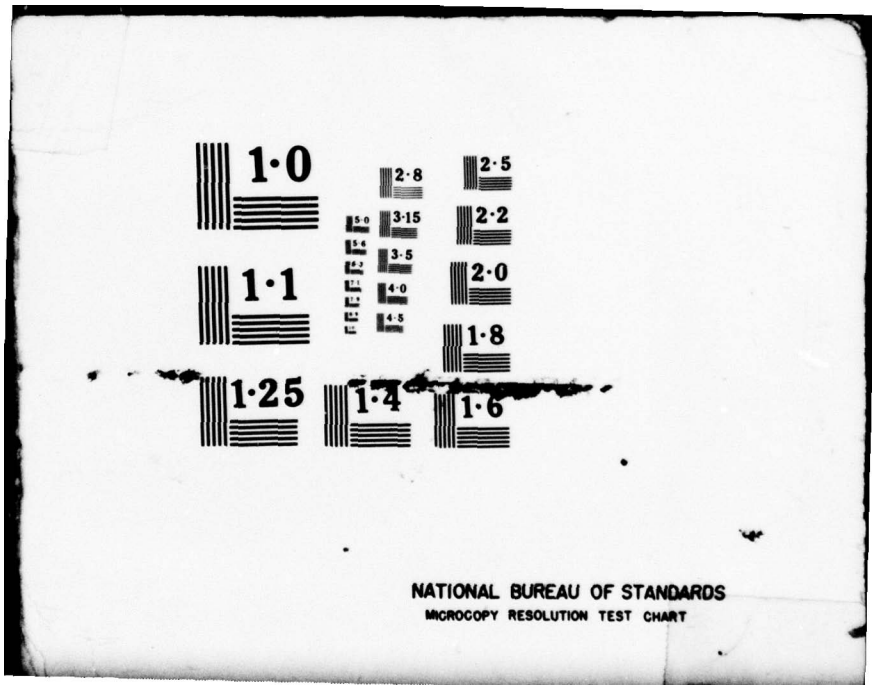
AD-A068 137 ARMSTRONG (PHILIP N) SANTA ANA CA F/G 9/2
COMPUTATION WITH SERIAL MEMORY SYSTEMS.(U)
FEB 79 P N ARMSTRONG, M REM N00014-78-C-0357
UNCLASSIFIED NL

1 OF 1
ADA
068137

END
DATE FILMED
6-79
DDC


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
5·0
2·5
2·2
2·0
1·8
1·4
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

code 437

LEVEL II

B.S.

(9) FINAL TECHNICAL REPORT, ON

(6) COMPUTATION WITH SERIAL MEMORY SYSTEMS.

Sponsored by
Office of Naval Research
Contract NOOOI4-78-C-Ø357
(15)

Contractor:
(10) Philip N. Armstrong, Martin Rem
17331 Keegan Way
Santa Ana, Ca. 92705

AD A068137

This is a report of an investigation made by Philip N. Armstrong in collaboration with

Martin Rem
Department of Mathematics
Eindhoven University of Technology
P. O. Box 513
Eindhoven
The Netherlands

(12) 64p.

(11) February 1979

DDC FILE COPY

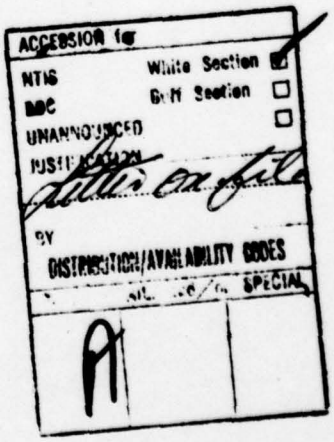
Preface

This is a report of an investigation which began at the Computer Science Department of the California Institute of Technology, where Martin Rem and I began a study of a special data sorting system applied to numerical computations. This study was done in part under contract with the Defense Advanced Research Projects Agency (DOD), ARPA Order Number 3314. The proposal for the investigation reported here was a result of this ARPA study.

Some of the preliminary ideas behind our study are contained in the book Sorting And Searching, Volume 3, by Donald E. Knuth, published by Addison-Wesley, 1973.

Dr. Rem returned to his home in Eindhoven, The Netherlands, before this report could be completed and hence cannot be held responsible for such errors as it may contain.

Philip N. Armstrong
February, 1979



INTRODUCTION

The question considered in this paper, and the motivation for the analysis which it contains, is: "Is it possible to substitute serial memory for conventional random access memory in commonly encountered numerical computations?" The reason for interest in this question is the current interest in such tasks as solving large systems of linear equations, large linear programming problems, and other calculations applied to large amounts of data, together with the current development of techniques for constructing large serial memories which are intrinsically less expensive than random access memories of comparable size and speed.

The above question may be given an affirmative answer for those computations discussed in this paper, if the serial memory is equipped with compare/exchange circuits so that it will sort its contents. Generally when using self-sorting memory, instead of conventional random access memory, labels or keys will be appended to the data to be stored; conventional machine addresses will not be used. The data entries thus specified will pass to simple arithmetic processors for the requisite numerical operations, after which the data may be returned to the self-sorting memory for retention until it is needed again. A property common to all of the computations discussed in this paper is the feasibility of computing the requisite memory output sequence as data is first passed to the memory, as well as in the course of the computation. While there is clearly no analytic basis for asserting that the class of all common numerical computations shares this property, it is clear that many computations not discussed in this paper are within this class.

The memory systems which are described in this paper are modules composed of serial registers, augmented with simple sort circuits. Any number of such modules may be combined at a cost which will be very nearly proportional to the number of modules, without significant

degradation of the memory performance.

The essentially serial nature of the memory suggests that instead of the access to many bits of a "word" which conventional random access memory provides, simultaneous access to corresponding bits of many different "words" or "records" would be appropriate. Such access, in turn, suggests that to a single memory module should correspond a processor so that simultaneous computations could be performed on many records. The outline of computing systems of this kind forms Part I of this paper. Part II contains descriptions of the application of the systems to three computations: Gaussian elimination with partial pivoting, formation of the transitive closure of a graph, and the simplex method of linear programming.

The principal conclusions to be drawn from Part II are:

1. Gaussian elimination
   Gaussian elimination with partial pivoting may be performed on an nxn matrix to obtain the L U decomposition in approximately $3/2n^2$ operations with n processors. This computation requires at least $n^3/3$ operations for a conventional computer.

   This same computation may be performed as the matrix is scanned, thus obtaining the factors after n^2+2n operations. This computation is not feasible with conventional computers.

2. The simplex method of linear programming
   A technique is displayed which permits a system of n variables with m constraints to be solved in an amount of time proportional to m+n. A conventional computer will require an amount of time (approximately) proportional to $nm+m^2$.

3. The transitive closure of a graph with n vertices
   The transitive closure of a graph of n vertices may be constructed from its incidence matrix in not more than n^2 operations. This computation requires n^3 operations for a conventional computer, for at least some graphs.

Other Computations

There are various incidental results which may be inferred from the contents of Part II. The systems which are described may be applied to various computations, including the few listed here:

1. Vector Transformations
   Facility for the manipulation required for the Gaussian elimination process, i.e. the performance of elementary matrix transformations, underlies many processes. Thus, for example, vector transformations, i.e. formation of the product Ab where A is an nxn matrix and b is a vector of n components. The product may be computed in 2n operations.

2. The Fourier Transform
   This is a special case of the above computation, where an entry a_{ij} of the matrix A is the complex entry $w^{(i-1)(j-1)}$ where w is a primitive n^{th} root of unity.

3. Matrix Multiplication
   The product AB of an nxn matrix A in System I and an nxn matrix B may be computed as the entries of B pass to System I in order by column. This procedure may be completed in n^2 operations.

4. Formation of the transpose of a large matrix
   This is a special sorting problem and as such is clearly appropriate for the sorting systems which are described in this paper. Efficient procedures for this computation have not yet been formulated, using the special systems described herein.

SELF-SORTING MEMORY

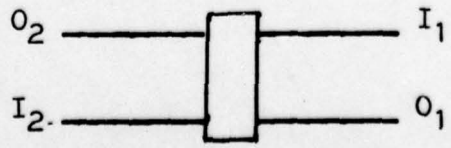
Special self-sorting memory (SSM) is discussed in some detail in the paper "A Serial Sorting Machine" by P. Armstrong and M. Rem. The functional properties of such memory which are of interest here will be briefly described.

SSM is composed of serial memory registers, e.g. MOS, CCD or magnetic bubble devices, which circulate data through simple circuits. The system accepts data for input via a sort circuit, and data will pass from a sort circuit to the system output. In the computations discussed in Section II of this paper, data will pass to an SSM module at the module input rate and may then be withdrawn in sorted order immediately after the data has been placed in the store. No delay is required for rearrangement within the SSM. If data is already in the store, a data record may pass into the store and thereby displace the next record in the output sequence. An SSM thus has two modes: it may accept data for input and storage, or it can accept data for displacement, so that as a record passes to it, the next record will simultaneously pass from it.

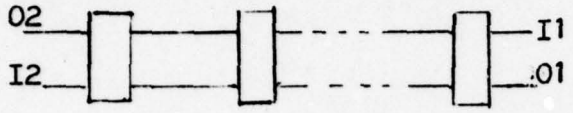
The "sorted order" mentioned above is simply the order prescribed by the magnitudes of the binary numerals which the data records represent. Data will pass to an SSM in the form of fixed length records, i.e. contiguous collections of some number n of bits. As data records pass through the SSM, they will pass to sort circuits so that corresponding bits of pairs of records will be compared and the first inequality of corresponding bits will determine which of two records will be treated as the greater one. Thus if a record is to be composed of a key which is followed by data, the key must pass to a sort circuit first, in decreasing order of the significance of its successive bits.

Modularity of SSM Systems

A single SSM module may be represented thus:

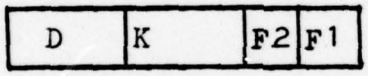


A record may be transmitted to an SSM module via line I2 for storage or for displacement; lines I1 and also O1 are used for connecting the modules and are discussed in the above paper. The output record will appear on the line O2, Systems composed of several modules may be systematically constructed by connecting modules:



Systems composed of several modules may be used for the computations of interest here without degradation of performance produced by their enlarged size.

In the computations to be discussed, a typical record will have the form indicated below, unless otherwise noted:



In the matrix computations which are discussed in Section II, the key K of a record will be the row index of the entry and the entry itself will be in the data part D of the record. Bits F1 and F2 are reserved to permit scanning of the contents of the SSM. For example, if a data record is passed to the SSM with a 1 in flag bit position F1, it will be treated as a record with a key larger than any record with a 0 in F1. Thus if the record output sequence begins with the record in the SSM with the smallest key, (the record with the smallest key will be called the smallest record) the smallest record will be displaced from the SSM by a larger record. In normal use, records will be transmitted to the SSM for storage and, after the records are in the SSM they will be displaced by records with a 1 in F1 or F2--or in both positions. The records which displace data from an SSM may very well have new keys so that they need not form a sequence which is even related in an obvious way to the sequence in which they were obtained from the SSM.

All of the record bits in selected bit positions may be changed to 0 or 1 within an SSM. This is accomplished by transmitting, via a single line, a signal to each SSM register. By properly adjusting the time of transmission of the signal, the selected bit positions may all be set to 1 or 0 within a single circulation of the SSM registers. This facet of SSM storage is useful in controlling flag bits: when all of the records within an SSM have their flag bits F1 or F2 set to 1, it is useful to set them to 0 so that new records may replace those in the store.

Interleaved Lists

In the computations described in Part II of this paper, it is presumed that to each sorted list there will correspond a unique SSM. Thus in matrix computations no two matrix columns will be contained in a single SSM. In some uses for SSM's a single SSM could contain several different ordered lists. If access to the first member (in the SSM the first member will be the one with the least key) of each list is required before access to the successor of a list member is permitted, the access time to the members of a list will be multiplied by the number of lists in the memory. To illustrate the idea, suppose two lists A and B are to be processed so that retrieval of a member of A should always be followed by retrieval of a member of B, and retrieval of a member of B should be followed by retrieval of a member of A. Suppose further that there are p members of A and q members of B. The members of A may be assigned keys which represent even numbers, and the keys assigned to members of B may be assigned odd numbered keys. Suppose $q < p$. Then if the largest key assigned to a record of A is 2p, and the largest key assigned to a member of B is 2q+1, a record from A retrieved from the SSM may be returned to it to retrieve the next member of B merely by setting flag bit F1 to 1. But a member of B which is retrieved from the SSM must be given a new key: it must be given the key k+2q mod 2p, if k is the key of the record from the SSM. If $k+2q > 2p$, flag bit F1 must be set to 1. When all of the mark bits have been set to 1, they must all be set to 0 so that the insertion/deletion operation may continue.

Of course, many lists may be processed in this manner in a single SSM and hence the number of modules in a system may be systematically decreased as the number of lists in a module is increased--but with correspondingly greater access time. In fact, just one SSM will suffice if enough time is allotted for processing the successive entries.

SSM Computing Systems

There are two configurations of SSM systems which are discussed in Part II of this paper. The first, System I, is composed of n identical modules, each of which consists of a single SSM and a processor. The second, System II, contains more communication facilities than System I, together with two special modules, Mc and MO.

System I

System I is composed of a linear sequence of n identical SSM/processor modules. Modules M1 and Mn accept data from and transmit data to exterior data input/output equipment. Modules M2, M3, ... , Mn-1 transmit data to and from their adjacent neighbors. In addition, any module may transmit data to the single system bus, and all of the modules may receive data from the bus.

Each module Mi is composed of a processor, Pi, and a module of SSM storage, SSMi. Pi may transmit a record to SSMi and may simultaneously retrieve a record from SSMi. All data which passes to or from SSMi must pass through Pi, from which it may be directed elsewhere, i.e. to the bus, to Pi+1, or to Pi-1--or back to SSMi.

System I is illustrated in Figure 1.

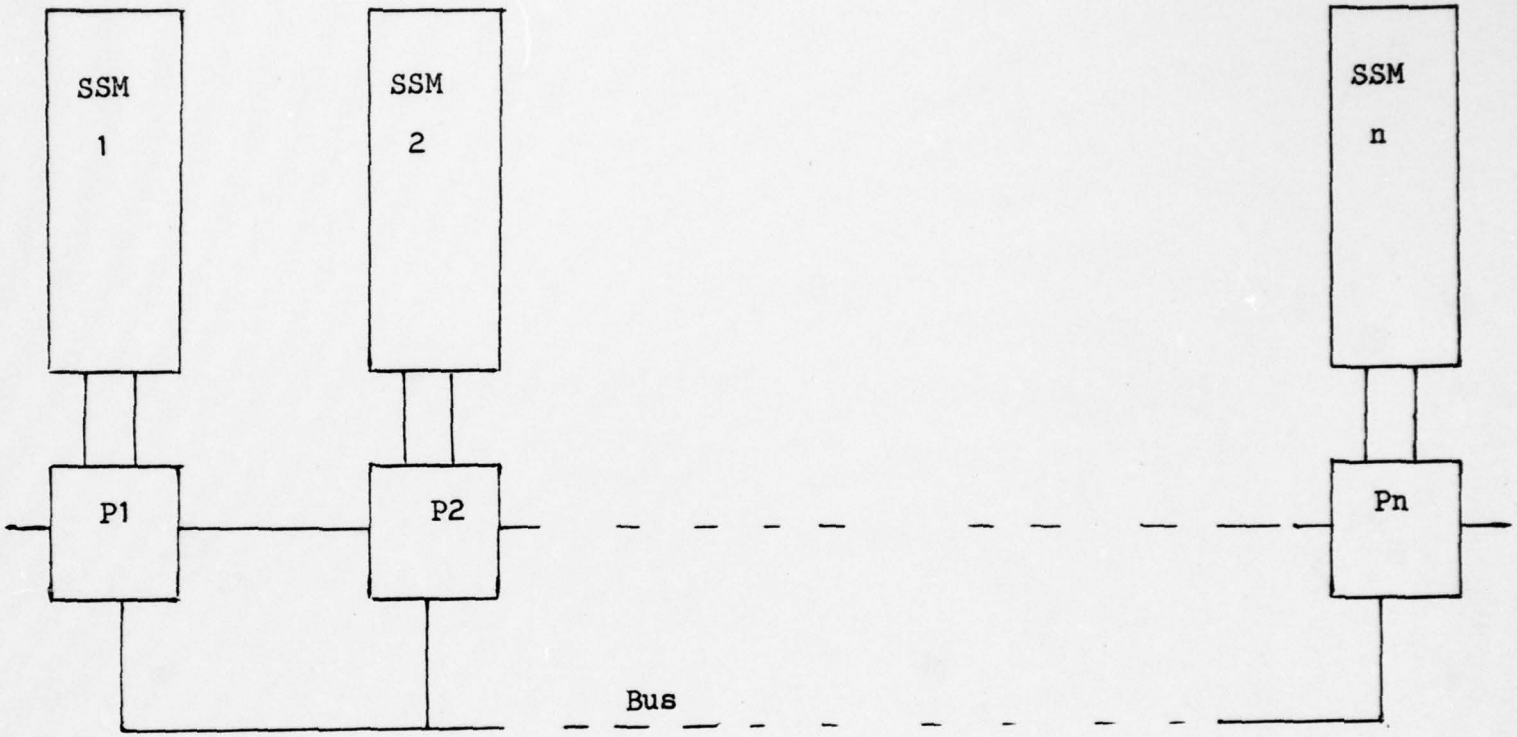
System II

System II has three busses, and two communication channels between adjacent processors. The components of System II are essentially similar to System I: the SSM systems are identical and the arithmetic processors are comparably simple. Any processor is permitted to transmit data on any of the three busses or to its adjacent neighbor via either of two channels.

System II contains two special modules, MO and control module Mc. These modules are special because they do not require the arithmetic facilities of the processors in the other modules.

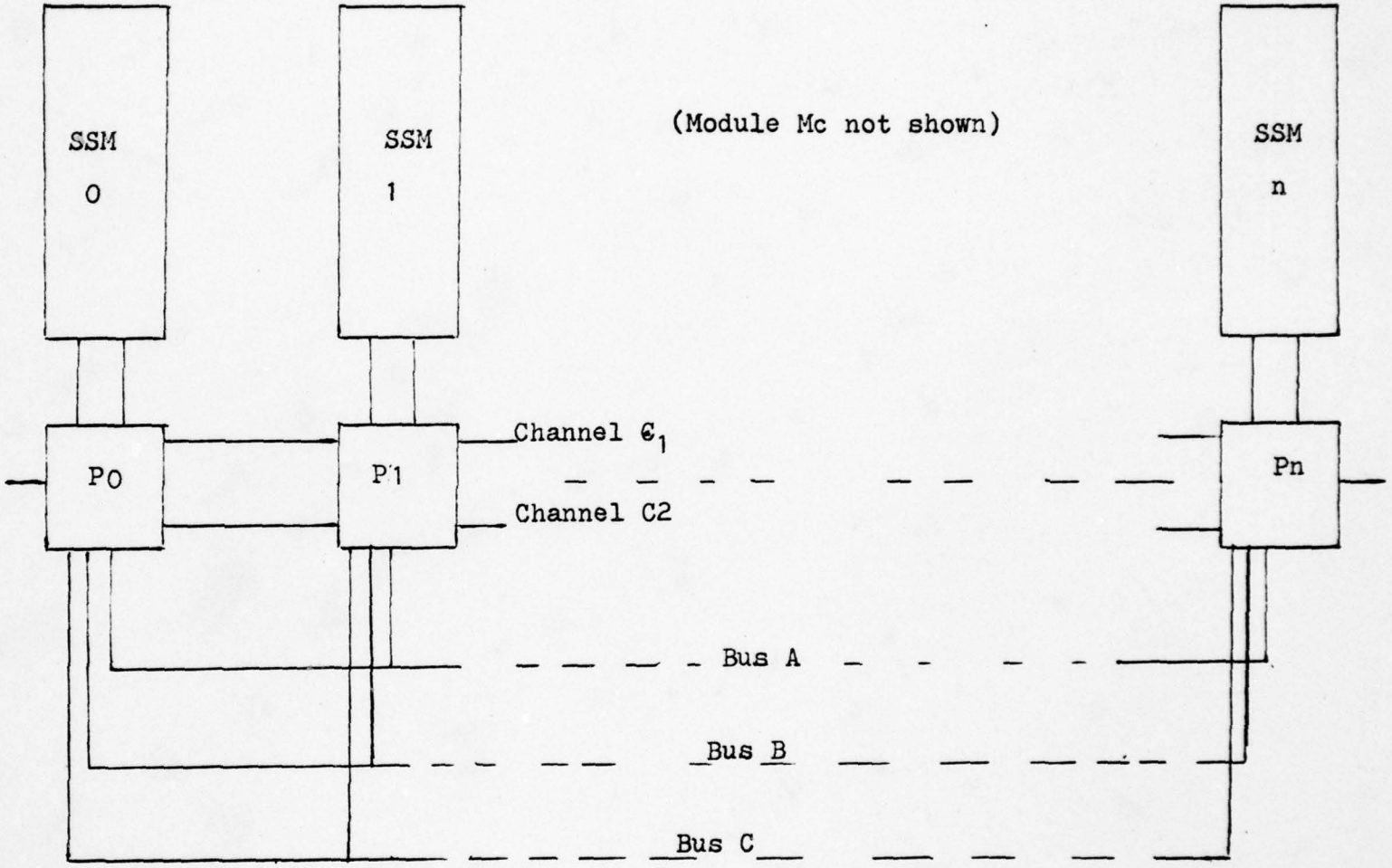
System II is illustrated in Figure 2.

The processors of Systems I and II require only facility for single serial arithmetic operations (they contain one multiplier and one adder) and adequate control to select the data paths for data to and from the lines to which they are connected.



SYSTEM I

Figure 1.



SYSTEM II

Figure 2

Module Mc

Control module Mc is designed to record the permutation which must be made before arithmetic computation may be performed on input data. Although the only applications for Mc which are described in Part II are to Gaussian elimination with partial pivoting, the utility of such a system appears to be general--it may be used to rearrange record sequences as this operation is needed for other computations.

Module Mc will contain one SSM unit, SSMc, and a processor Pc. The module SSMc will contain the list of the assigned indices, sorted to the input record sequence. A record in SSMc will be a pair (i,j) where i is the serial number in the input sequence at which the record will appear which should receive the key j, where j may be arbitrarily chosen.

In the procedures of interest, the memory will first contain records with pairs of identical entries (i,i), ($1 \leq i \leq n$). When the p_jth record is found to be the new pivotal record, so that the p_jth record should be assigned the key j, the number p_j may be retained in Pc and as SSMc is scanned, the data part of each non-pivotal record may be changed from k to k+1 until record p_j is reached ($i \leq k < p_j$). The record p_j will then be assigned the key j, i.e. the record in SSMc with key p_j will be given the data part j, which will also become the key of the input record to MO which contains the entry in the p_jth row of the input column.

PART II

The following computations are described as they would be performed with Systems I and I:

1. Gaussian Elimination With Partial Pivoting

2. The Simplex Method of Linear Programming

3. The Transitive Closure of A Directed Graph

GAUSSIAN ELIMINATION

Let A be an invertible nxn matrix and b a vector with n components. The system Ax = b may be solved by the following procedure:

1. Factor A to obtain factors P, L, and U where PAx = LUx = Pb.
   Here the nxn matrix P is a permutation matrix.
   L is a lower triangular matrix with 1's on its diagonal.
   U is an upper triangular matrix.
2. Solve the system Ly = Pb for the vector y.
3. Solve the system Ux = y.

If the factors L and U are obtained by systematically selecting the entry of greatest magnitude in a column for placement on the diagonal of U, the method is called Gaussian elimination with partial pivoting, and is the one to be described. Nearly every modern publication on numerical analysis will contain mention of computing the matrices P, L and U. See, for example, the book Numerical Analysis by Germund Dahlquist and Ake Bjorck, published by Prentice Hall, 1974.

In the first part of the discussion, it is presumed that operations on all n entries of a row may be performed. It will then be shown how the procedure may be applied as the n^2 terms of the matrix A pass to the computing system serially.

Construction of Matrices L and U

The matrices L and U are obtained by constructing the sequence of nxn matrices $A_1, A_2, \ldots, A_k, \ldots A_n$ from the matrix A according to the relations defined below. (The entries of matrix A_k are the terms a^k_{ij}; p_k is the row index of the pivot row in matrix A_k.)

1. $a^k_{p_k k} = \max(a^k_{ik}) \quad (k \le i \le n)$
   ($\max(a_{i,j})$ is the entry of greatest magnitude over the indicated range)
2. $a^{k+1}_{i+1,j} = a^k_{ij} - \dfrac{a^k_{ik}\, a^k_{p_k j}}{a^k_{p_k k}} \qquad (k \le i < p_k,\ k < j \le n)$
3. $a^{k+1}_{rj} = a^k_{p_k j} \qquad (r = n-k+1,\ k \le j;\ r = k,\ j < k)$

The entries of U are stored with the rows in reversed order in preparation for the back substitution procedure.

4. $a_{ij}^{k+1} = a_{ij}^{k} - a_{ik}^{k} a_{p_k j}^{k}$ $(p_k < i \leq n)$

5. $a_{rk}^{k+1} = a_{ik}^{k} / a_{p_k k}^{k}$ $(r=i+1,\ k \leq i < p_k;\ r=i \text{ for } p_k < i \leq n)$.

The parts of the matrix to which these various relations apply are illustrated in the drawings of Figure 1, on the page following. The numbering of the drawings corresponds to the numbering used here for the equations.

The Matrix P

The matrix P is completely determined by the entries p_k $(1 \leq k \leq n)$ in equation 1. It is not explicitly constructed in the computation: it is sufficient to retain the indices p_k in processors Pk.

Mechanization

System I is suited to this computation.

To a column j of a matrix A we associate a module Mj of System I so that the entries of the jth column will be stored in SSMj. The records which contain entries of row i will be assigned the key i in each module. In the computation, an entire row of a matrix will be available at once. In a typical operation, the record with the least key and with 0 in flag bit positions F1 and F2, will be displaced from SSMj by a record with a 1 in F1 or F2 transmitted from Pj. When all of the records with 0 in F1 or F2 have been displaced from SSMj, the flag bits F2 of all of the records in SSMj will be set to 0. This procedure will cause the records with 0 in their respective flag bit positions F1 and F2 to pass to the SSMj output as records with either F1 or F2 set to 1 are passed from Pj to SSMj. Those records with 1 in F1 will be retained in SSMj because they will not be displaced by records with flag bit F2=1 if F1=0. These records will contain the entries of the matrix U. Each time it is found that all of the records

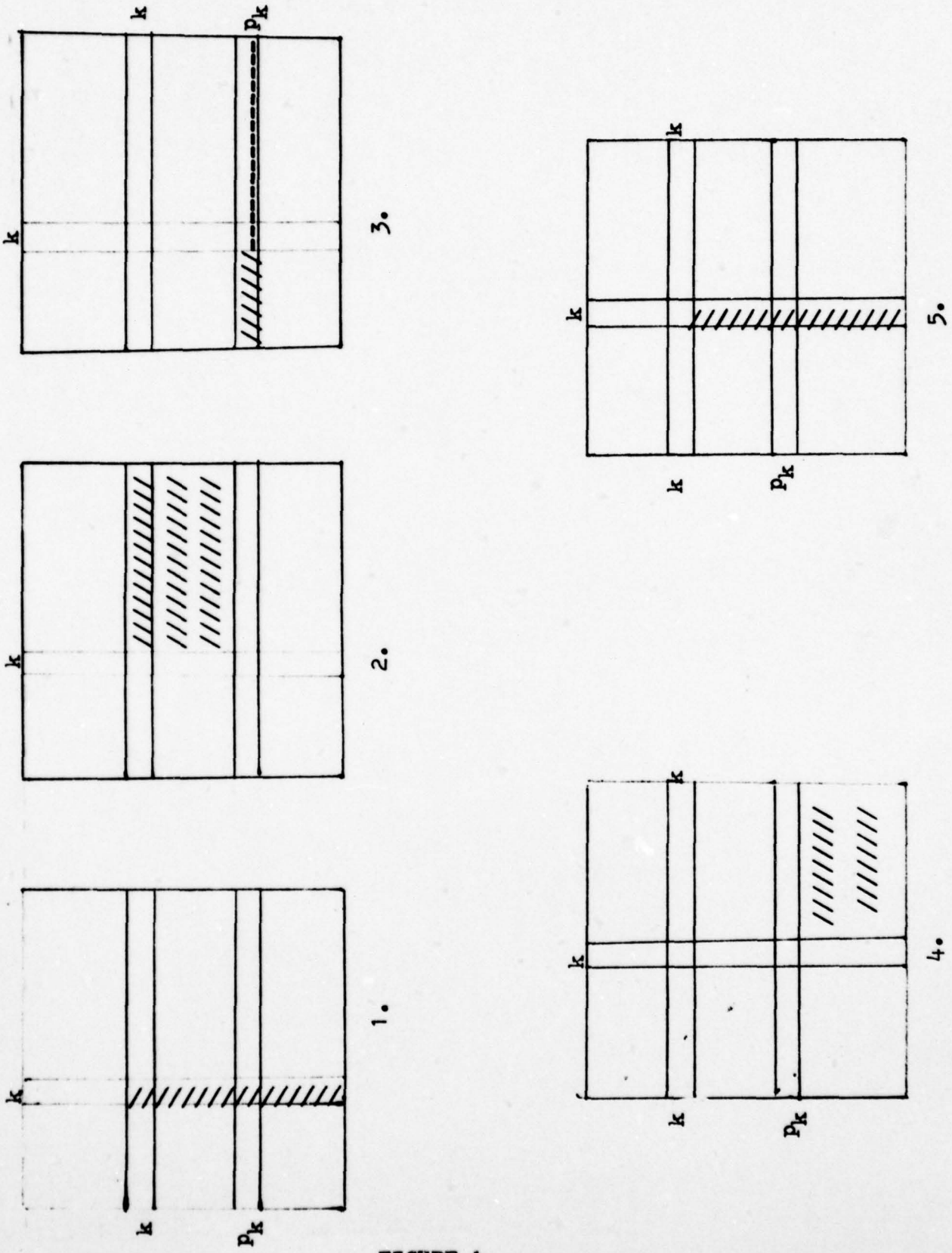


FIGURE 1

have 1 in F1 or F2, the computation of a matrix in the sequence A_1, A_2, ...A_k, ... , A_n will have been completed. When all of the records in module Mn have F1=1, the computation of A_n will be complete.

The Input Process

The matrix A may be in place after other computation in System I; otherwise transmission of A will be required before the computation may begin. If transmission of A is required, the first pivot selection may be made as data passes to System I.

The entries of the matrix A will pass to processor P1 serially by column, beginning with the last column. Within a column, the entries are presumed to pass in increasing order of row indices. Thus a_{ij} will precede a_{kt} if $t < j$ and a_{ij} will precede a_{kj} if $i < k$.

When an entry a_{ij} $(1 < j < n)$ passes to P1, the flag bits F1 and F2 of the record in which it is contained will be set to 0. The record will be passed to P2, P3, ... , Pj without alteration. In Pj the first two entries a_{1j} and a_{2j} will be retained in Pj; the retained in Pj; the remaining entries of column j will pass to SSMj in records with the row indices for keys. When all of the entries of column j have passed to module Mj, they will be retained in Pj or SSMj until column 1 passes to P1; SSMj will then contain the $n-2$ entries a_{ij} $(3 < i < n)$. Entry a_{1j} and a_{2j} will be retained in Pj.

At the beginning of the transmission of column 1, the first two entries will be compared and the smaller (the entry of smaller magnitude) one will be transmitted to SSM1. Similarly as the successive records containing entries of column 1 pass to P1, a signal will be transmitted to the bus if a record in P1 is exchanged for a record which is received from the input to P1. This procedure will continue in each processor until the last column entry passes to P1.

Notice that each SSMi $(1 \leq i)$ contains n+2 entries. As an entry is transmitted to SSMi from Pi during the column 1 input process a record will be displaced from SSMi.

When the last entry of column 1 has passed to P1, it will be compared with the entry retained in P1 so that the pivot row may be identified and each processor Pj will contain the entry $a_{p_1 j}$. All of the records in SSMj ($1 \leq j \leq n$) will then have their F2 bits set to 0 and processor P1 will transmit $a_{p_1 1}$ to the bus so that each processor Pj will have $a_{p_1 1}$ and $a_{p_1 j}$ ($1 \leq j \leq n$) in its store, as well as one non-pivotal entry. It is convenient to presume that the non-pivotal row will be transmitted to the SSMs for storage before the computation begins so that each SSM will then contain n-1 entries and each processor Pj ($2 \leq j \leq n$) will contain two entries, $a_{p_1 1}$, and $a_{p_1 j}$. Processor P1 will contain $a_{p_1 1}$.

If this procedure is not followed, the first column must be scanned for $a_{p_1 1}$ before the computation may begin--and, of course the remaining columns must concurrently pass to their respective processors for selection of the pivot entries $a_{p_1 j}$. This scan will require time for n operations.

The Computation

The computation will begin with the simultaneous transmission from all of the processors Pj ($1 \leq j \leq n$) of the entry $a_{p_1 j}$ to SSMj with flag bit F1 set to 1, thus displacing records containing the first non-pivotal row. In processor P1, the arithmetic computation which is performed will be that defined by equation 5; in processors Pj ($2 \leq j \leq n$), the arithmetic computation is defined by equations 2) and 4). Processor P2 will select the pivot $a_{p_2 2}$ according to equation 1). The pivot row p_2 will then be transmitted to the SSMs with flag bit F1 set to 1, thereby beginning the next iteration. Generally in the computation of matrix A_k, processors P1, P2, ... , Pk-1 will compute indices according to equation 5), and processors Pj ($k \leq j \leq n$) will compute entries according to equations 2) and 4). Processor Pk+1 will also perform the "computations" defined by equation 1): processor Pk+1 will select the next pivotal row.

Remark

If an entry of a pivotal row is zero, i.e. if for some k and j the entry $a^k_{p_k j} = 0$ in equations 2) and 4), Pj may be idle instead of performing arithmetic computation, although it must be used

in selecting the next pivot row.

Timing

The procedure will require operations on n-k rows to compute A_k, if the matrix is in place at the beginning of the process. There will thus be n(n+1)/2 operations required to obtain matrix A_n, after the input process is complete. Since the input process requires n^2 operations, the total number of operations may be seen to be n(n+1)/2 or $n^2+n(n-1)/2$, depending upon whether the input process must be performed.

Gaussian Elimination With Partial Pivoting On The Fly

The procedure for obtaining the matrices L and U may be performed as the entries of the matrix A are transmitted to System II. The computation itself will first be described and then its mechanization. In the description of the computation, it is convenient to describe the computed result in nearly the way in which the computation is mechanized and then describe some minor deviations from the procedure.

The Computation

The sequence of matrices $A_1, A_2, \ldots, A_n$ is computed from the nxn matrix A. Matrix A_k $(1 \leq k \leq n)$ in this sequence is an n by k matrix. It contains column k which is computed from column k of the matrix A and selected entries from A_{k-1}. Matrix A_k also contains the entries of A_{k-1}, but arranged to conform to the pivot operations performed on column k of A.

The first operation which is performed on the entries of column k is to rearrange these entries to conform to the pivoting operations which produced matrix A_{k-1}: the entries in column k which are members of the first k-1 pivotal rows are placed in the first k-1 positions of column k.

The entries of column k which have thus been rearranged will be denoted a_{ik}^{0*} $(1 \leq i \leq n)$. (The entries of column k of the matrix A will be denoted a_{ik}.) In computing the entries a_{ik}^{k} of column k, the entries a_{ik}^{k-1*} are first computed, where the entry a_{ik}^{k-1*} is given by

$$1.\quad a_{ik}^{k-1*} = a_{ik}^{0*} - \sum_{r=1}^{q} a_{ir}^{k-1}\, a_{rk}^{r-1*} \qquad (q < \min(i,k)).$$

An entry a_{ik}^{j*} $(j < k)$ is defined thus:

$$2.\quad a_{ik}^{j*} = a_{ik}^{j-1*} - a_{ij-1}^{k-1}\, a_{j-1,k}^{j-1*} \qquad (j-1 < \min(i,k))$$

3. $a_{ik}^{j*} = a_{ik}^{j-1*}$ $(\min(i,k) \quad j-1)$.

The pivot entry $a_{p_k k}^{k-1*}$ is defined by the relation

4. $a_{p_k k}^{k-1*} = \max(a_{ik}^{k-1*})$ $(k \leq i \leq n)$.

The entries a_{ik}^{0*} may now be defined:

5. $a_{i1}^{0*} = a_{i1}$ $(1 \leq i \leq n)$

6. $a_{ik}^{0*} = a_{p_i k}$ $(1 \leq i \leq k-1)$

7. $a_{rk}^{0*} = a_{ik}$ $(r=i+s(i),\ i \neq p_i)$

(Here $s(i)$ is the number of pivotal indices p_i greater than the index i, where i is not a pivotal index.)

The operations required to obtain column k, i.e. the entries a_{ik}^{k-1*}, are thus defined: the pivot entries are placed in the first positions of column k of A and then the arithmetic computations needed to obtain the entries a_{ik}^{k-1*} are performed. Finally the pivot entry $a_{p_k k}^{k-1*}$ is identified. The remaining procedures needed to obtain the entries a_{ik}^{k} are 1) rearrange the entries of column k to place a_{ik}^{k-1*} in the diagonal position k of the column, and 2) compute the entries of L, i.e. those for which, after the rearrangement, $i < k$. These operations are characterized by the following relations:

8. $a_{ik}^{k} = a_{ik}^{k-1*}$ $(1 \leq i \leq k-1)$

9. $a_{kk}^{k} = a_{p_k k}^{k-1*}$

10. $a_{rk}^{k} = a_{ik}^{k-1*} / a_{kk}^{k}$ $(r=i+1,\ k \leq i < p_k;\ r=i,\ p_k < i \leq n)$.

The matrix A_k will be formed when the entries of A_{k-1} have been rearranged to conform to the placement of $a_{p_k k}^{k-1*}$ in the kth position of column k:

11. $a_{ij}^{k} = a_{ij}^{k-1}$ $(1 \leq i \leq k-1,\ 1 \leq j \leq k-1)$

12. $a_{kj}^{k} = a_{p_k j}^{k-1}$ $(1 \leq j \leq k-1)$

13. $a_{rj}^{k} = a_{ij}^{k-1}$ $(r=i+1,\ k \leq i < p_k,\ r=i,\ p_k < i \leq n;\ 1 \leq j \leq k-1)$

Deviations From The Procedure

The permutation by which a_{ik}^{0*} $(5 \leq k)$ is formed will place the first k-3 pivot entries in the first k-3 positions of column k. The entry $a_{p_{k-2}k}$ will be labeled so that after the entries a_{ik}^{k-3*} have been computed, the entry $a_{p_{k-2}k}^{k-3*}$ will be placed in the k-2nd column position. The entry $a_{p_{k-1}k}^{k-3*}$ will be selected so that from the entry $a_{p_{k-1}k}^{k-3*}$ and the entry $a_{p_{k-2}}^{k-3*}$, the entry $a_{p_{k-1}k}^{k-2*}$ may be be computed:

$$a_{p_{k-1}k}^{k-2*} = a_{p_{k-1}k}^{k-3*} - a_{p_{k-1}k-2}^{k-1}\ a_{k-2,k}^{k-3*}$$

The entry $a_{p_{k-1}k}^{k-2*}$ may then be used in relation 2) to compute a_{ik}^{k-1*}.

The entries a_{ik}^{k} for which $i \leq k$ will be assigned keys so that they will appear in reversed order for the back substitution process: the entries of U will be assigned the keys n-i+1.

The procedure for computing the entries a_{ik}^{k} for $k \leq 5$ will be described for each value of k in the next section.

Mechanization

System II is intended for this computation. Reference to its description in Part I is suggested.

The entries of A will pass to module M0 in order by column and, within a column, in order by row. The entries of column j ($1 \leq j \leq n$) will first pass to processor P0 in module M0 for assignment of record keys, and then to SSM0 for arrangement of the column entries. The entries will then pass to processors P1, P2, ... , Pj, for arithmetic computation and storage in module Mj.

As the entries pass to SSMj ($0 \leq j \leq n$), the first n-1 entries will pass to the system set to its input mode. When the last entry passes to SSMj, the system will be set to its displacement mode so that generally the last column entry will displace a record from the SSM to which it is transmitted. If the entries in SSMj have a flag bit F1 or F2 set to 1 and transmission of a record with a flag bit set to 1 thereby displaces a record with a flag bit set to 1, the flag bits F1 or F2 will be set to 0. This procedure makes possible repeated displacement of records from an SSM by new ones.

More of the details are contained in what follows; it is convenient first to describe the passage of the first four columns, and then describe the passage of column j ($5 \leq j \leq n$). Reference to the indicated figures is suggested.

Column 1

The entry a_{i1} ($1 \leq i \leq n$) will pass to P0 and will be assigned the key i. The entry will then pass to SSM0 with with flag bit F1 set to 1. When the first n-1 records have passed to SSM0, SSM0 will be set to its displacement mode and will remain with this setting until the end of the back substitution procedure (described in the next section of this paper).

When the last record, with entry a_{n1}, passes to SSM0 it will

displace the record with entry a_{11}^{0*}. This displaced record will have F1 set to 1, with the result that all of the F1 bits in SSM0 will be then set to 0. The entries a_{i1}^{0*} will pass to P1 for selection of $a_{p_1 1}^{0*}$. As the entries a_{i1}^{0*} pass to P1 they will be compared and the one of greatest magnitude will be retained in P1. The remaining n-1 entries will pass to SSM1 with flag bit F2 set to 1 and with F1=0. The entry $a_{p_1 1}$ will then be placed in a record with the key n and will be transmitted to SSM1 with F1=1. (The entries a_{ij}^{j} of the matrix U will be assigned the keys n-i+1 and flag bit F1 will be set to 1 for such records.) The entry $a_{p_1 1}^{0*}$ will also be retained in P1 for use in later computations. The record with a_{n1}^{1} will displace a record with F2=1, and hence all of the F2 bits in SSM1 will be set to 0. The index p_1 will be transmitted to processor Pc in control module Mc as a_{p1}^{0*} is transmitted to SSM1.

Column 2

The entries a_{i2} will pass to module M0 as the entries a_{i1} pass from it. The entry a_{i2} will be assigned the key i and flag bit F2 will be set to 1 for each record--again the record with a_{n2} will displace the record with a_{21}^{0*}, with flag bit F2=1. The records in SSM0 will then have all of their flag bits set to 0 and the process will continue as the entries a_{i3} pass to SSM0 and the entries of column 2 pass via processor P1 to processor P2. As the entries a_{i2} pass through P1, P1 will compute $a_{i1}^{1}=a_{i1}^{0*}/a_{11}^{1}$ $(1 < i \leq n)$. The entry a_{i1}^{1} will be placed in a record with F1=0, F2=1, and with its key set to i+1 $(1 \leq i < p_1)$ or to i $(p_1 < i \leq n)$. The entry $a_{p_1 2}$ will be labeled (F1 will be set to 1) in module M0 as a_{i2}^{0*} passes from M0. The record with this entry will have its key set to n and the record with this entry will then pass to SSM2 with F1=1. The last entry of column 2 which is not a pivotal entry, i.e. a_{n2} if $p_1 \neq n$, will be retained in P2 so that there will be n-1 entries in SSM2 when all of the second column entries have passed to module M2.

The passage of the first two columns is illustrated, for completeness, in figures 2 and 3.

Column 3

The passage of column 3 to SSM3 is illustrated in Figure 4.

As the entries of column 3 pass to MO, the entry $a^{0*}_{p_1 3}$ will be assigned the key 1 and the entries a_{i3} $(1 \leq i < p_1)$ will be assigned the key i+1. The entries of column 3 will pass to SSMO as the entries of column 2 pass from SSMO to P1 and then to P2.

As the entries a^{0*}_{i3} of column 3 pass to P1, the entries a^{1}_{i1} will pass to P1 from SSM1. The first non-pivot entry a^{1}_{21} will pass to P1 as the record with the entry a^{0*}_{23} passes to P1.

The entry a^{0*}_{13} will be retained in P1 so that the entries

$$a^{1*}_{i3} = a^{0*}_{i3} - a^{1}_{11}\, a^{0*}_{13} \qquad (1 \quad i \quad n)$$

may be computed.

The entries a^{1}_{i1} $(1 < i \leq n)$ will pass to processor P2 for computation of

$$a^{1*}_{i2} = a^{0*}_{i2} - a^{1}_{i1}\, a^{0*}_{i2} \qquad (1 \quad i \quad n)$$

at the same time the entries a^{1*}_{i3} are computed in P1.

Column 4 (Figure 5)

The entries a^{0*}_{i4} of column 4 will be arranged so that $a^{0*}_{14} = a_{p_1 4}$ and as the entries a^{0*}_{i4} pass from module MO, the entry $a^{0*}_{p_2 4}$ will be labeled.

The entries a^{0*}_{i4} $(1 \leq i \leq n)$ will first pass to P1 for computation of a^{1*}_{i4} and then transmission to P4 for rearrangement so that $a^{1*}_{p_2 4}$ may be placed in the second column position. Notice that as the labeled record with $a^{0*}_{p_2 4}$ passes to P1, the entry $a^{1}_{p_2 1}$ may be identified (it will pass from SSM1 at the same time) so that it may be assigned the key n-1, and the keys of the entries (i.e. the keys of

records with the entries) a^1_{i1} for $2 \leq i < p_2$ will be increased by 1. The entries a^1_{i1} for which $p_2 < i \leq n$ will remain unchanged.

The entries a^2_{i2} will be computed in processor P2 and will then be passed to P3 for computation of the entries a^{2*}_{i3} and identification of $a^{2*}_{p_2 3}$.

Column 5

The passage of column 5 to P5 is illustrated in Figure 6.

As the entries a^{0*}_{i5} of column 5 pass to P1 from M0, $a^{0*}_{p_1 5}$ will be in the first column position and $a^{0*}_{p_2 5}$ will be in the second. Entry $a_{p_3 5}$ will be labeled (in a record with F1=1). The entries a^{1*}_{i5} will pass to P2 and from P2 directly to P5 simultaneously with the entries a^2_{i2} for computation of the entries a^{2*}_{i5} and storage of the entries of column 5 in module M5. The entries a^{1*}_{i4} will simultaneously pass to P2 for computation of a^{2*}_{i4} and will then pass via P3 to P4 with the entries a^3_{i3} for computation of the entries a^{3*}_{i4} and identification of the entry $a^{3*}_{p_4 4}$.

The computation of a^{3*}_{i4} is, of course, defined by the relation

$$a^{3*}_{i4} = a^{2*}_{i4} - a^3_{i3}\, a^{2*}_{p_3 4}.$$

The record with the entry $a^{1*}_{p_3 4}$ may be identified as the entries of column 4 pass to P4 from P1. Thus the entry $a_{p_3 4}$ may be passed to P2 before the remaining column entries. But computation of $a^{2*}_{p_3 4}$ requires access to the entries $a^{1*}_{p_2 4}$ and a^2_{42}:

$$a^{2*}_{p_3 4} = a^{1*}_{p_3 4} - a^{1*}_{p_2 4}\, a^2_{42}$$

The entry $a^{2*}_{p_3 4}$ may thus be computed from $a^{1*}_{p_3 4}$ if in P2 the entries

$a_{p_2 4}$ and a^{2}_{42} are accessible. The entry a^{2}_{42} may be retained in P2 from the previous column scan. Entry $a^{1*}_{p_2 4}$ will be labeled in MO and may therefore be selected by P4 as the entry passes to P4 from P1.

When all of the entries of column 4 have passed to P4, the entries $a^{1*}_{p_2 4}$ and $a^{1*}_{p_3 4}$ may be transmitted to P2 for computation of $a^{2*}_{p_3 4}$. This entry may then pass to P4 ahead of the remaining column entries to permit computation of the entries of column 4 in P4:

$$a^{3*}_{i4} = a^{2*}_{i4} - a^{2*}_{p_3 4}\, a^{3}_{i3}$$

When in the computation the entry $a^{2*}_{p_3 4}$ is identified in the sequence from P2, it may be effectively erased and the duplicate entries thereby removed.

During the period the entries a^{3*}_{i4} are computed in P4, P3 will be busy computing a^{3}_{i3}. It is for this reason that P4 is used for the computation of the entries a^{3*}_{i4}.

The configuration shown in Figure 7 is the one which will characterize the flow of data for columns j ($5 \leq j \leq n$).

The first j-3 entries of pivotal rows will be placed in the first j-3 positions of column j in module MO. The entry $a^{0*}_{p_{j-2} j}$ of the pivot row p_{j-2} will be labeled as the entry passes from MO to P1, P2, ... , Pj-3. The entries a^{j-4*}_{ij} and a^{j-3}_{ij} will then pass to Pj-2 for computation of the entry $a^{j-2*}_{p_{j-1} j} = a^{j-1*}_{p_{j-1} j}$ so that after compu-

tation in Pj-2 of the entries a_{ij}^{j-2*}, the entries a_{ij}^{j-1*} may be computed in Pj.

In each processor Pi ($1 \leq i \leq j-3$), the index p_{j-2} will be changed to j-2 when $a_{p_{j-2}j}^{j-3*}$ passes to Pi. The record keys of entries a_{ij}^{j-3} will be increased by 1 if $j-2 \leq i < p_{j-2}$ and will otherwise be unchanged.

The assignment of data paths is somewhat arbitrary in this description. For example, the entries a_{ij}^{j-4*} may pass from Pj-4 to Pj at the same time a_{ij}^{j-3} passes from Pj-3 to Pj.

The busses A, B and C need not provide direct communication to all of the processors in this particular computation. Inspection of figures 7 or 8 will show that it will be sufficient if each processor Pj has two channels to each of the processors $Pj\pm1$, one channel each to the processors $Pj\pm2$ $Pj\pm3$ and one channel to each of the processors $Pj\pm4$. The data path to module Mc may be via the unused channels from Pj-1 to modules MO and Mc.

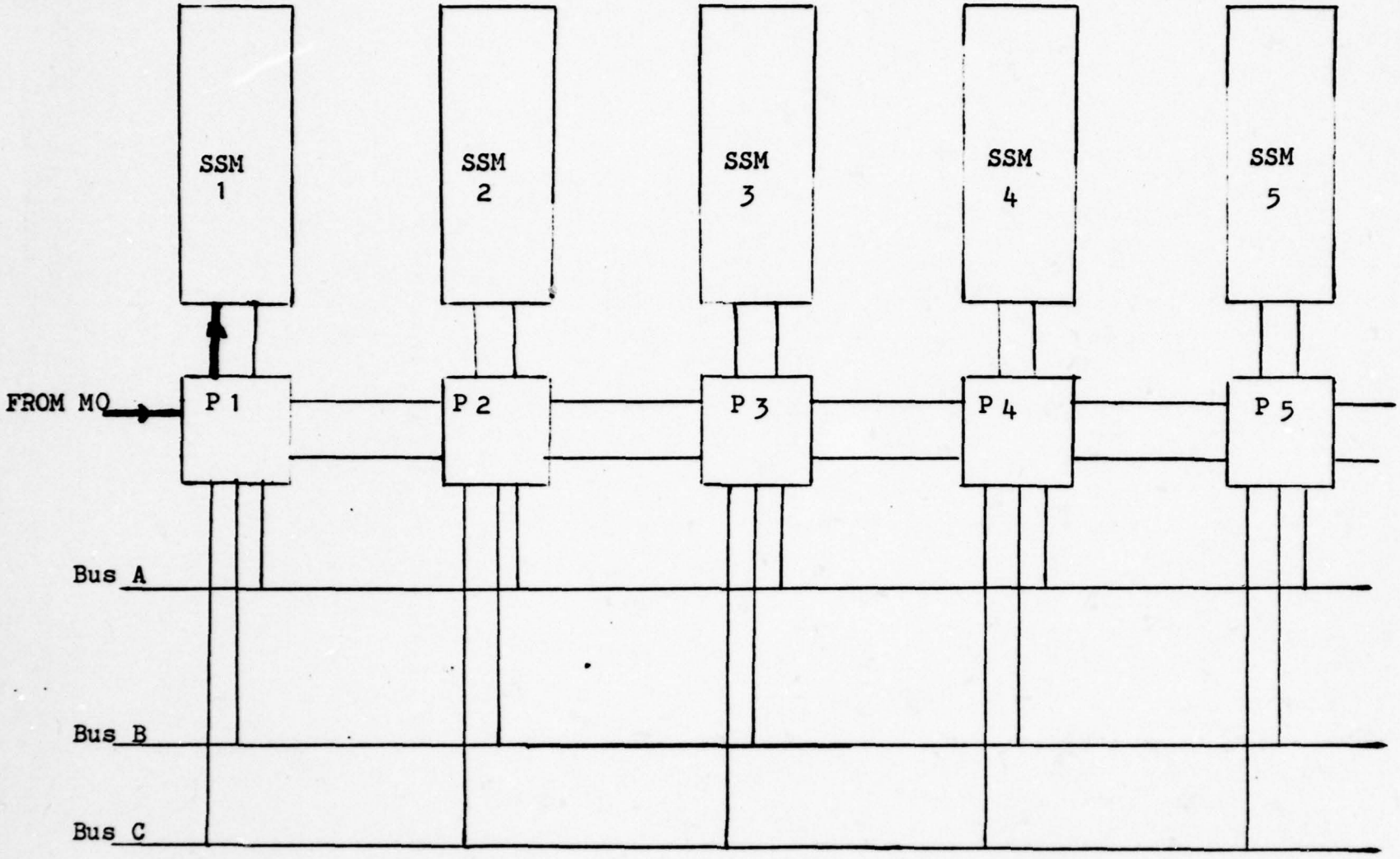


FIGURE 2

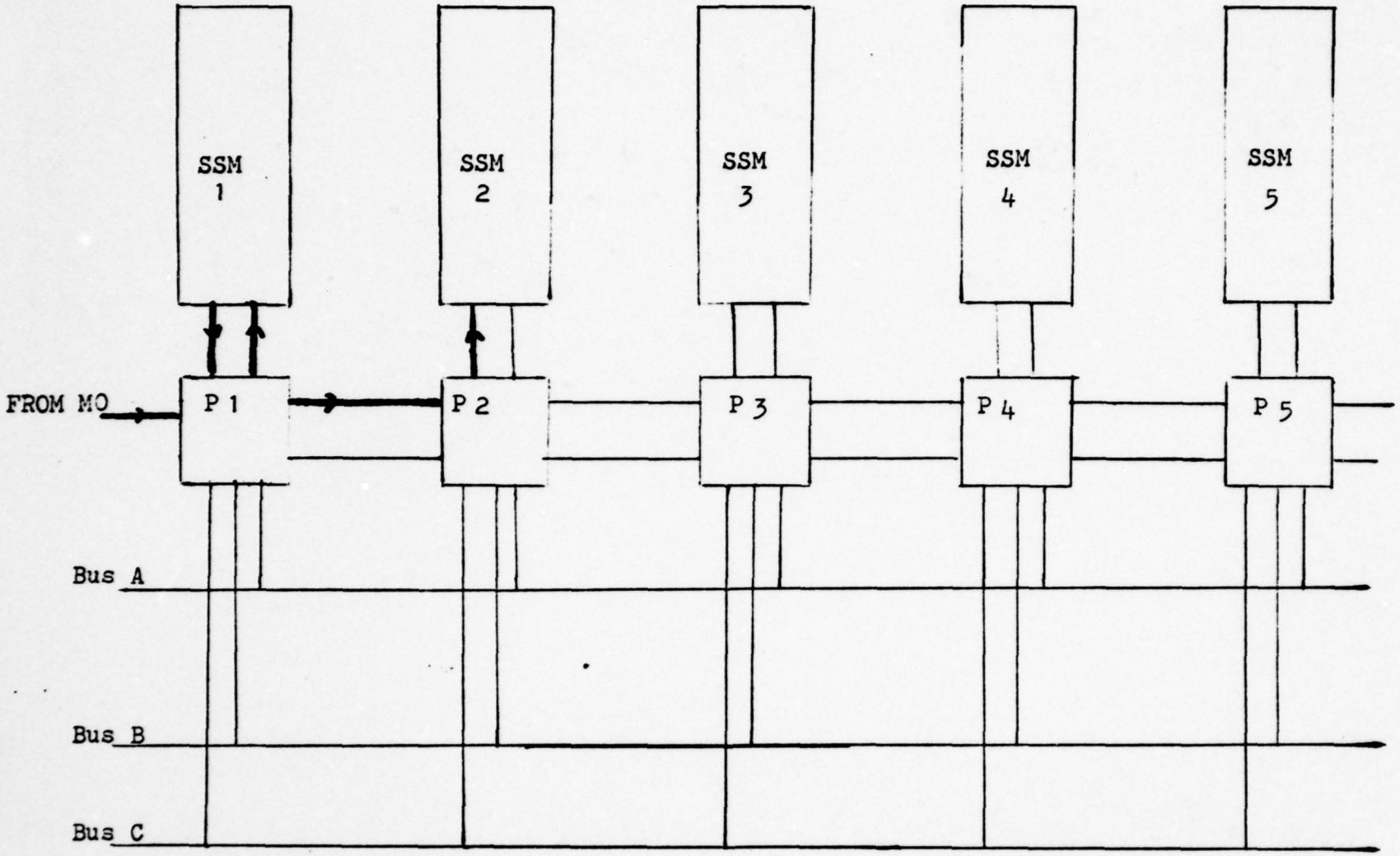


FIGURE 3

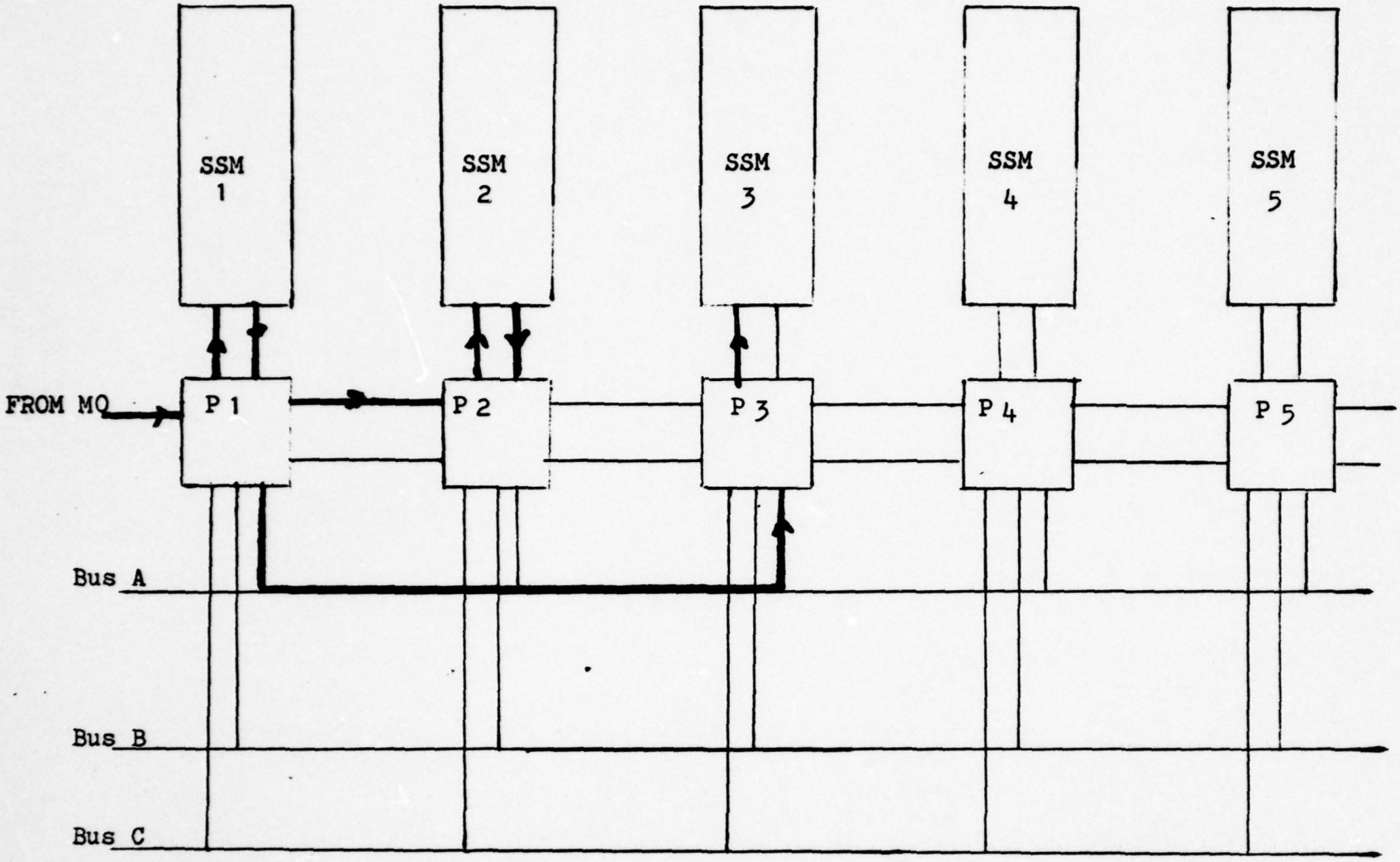


FIGURE 4

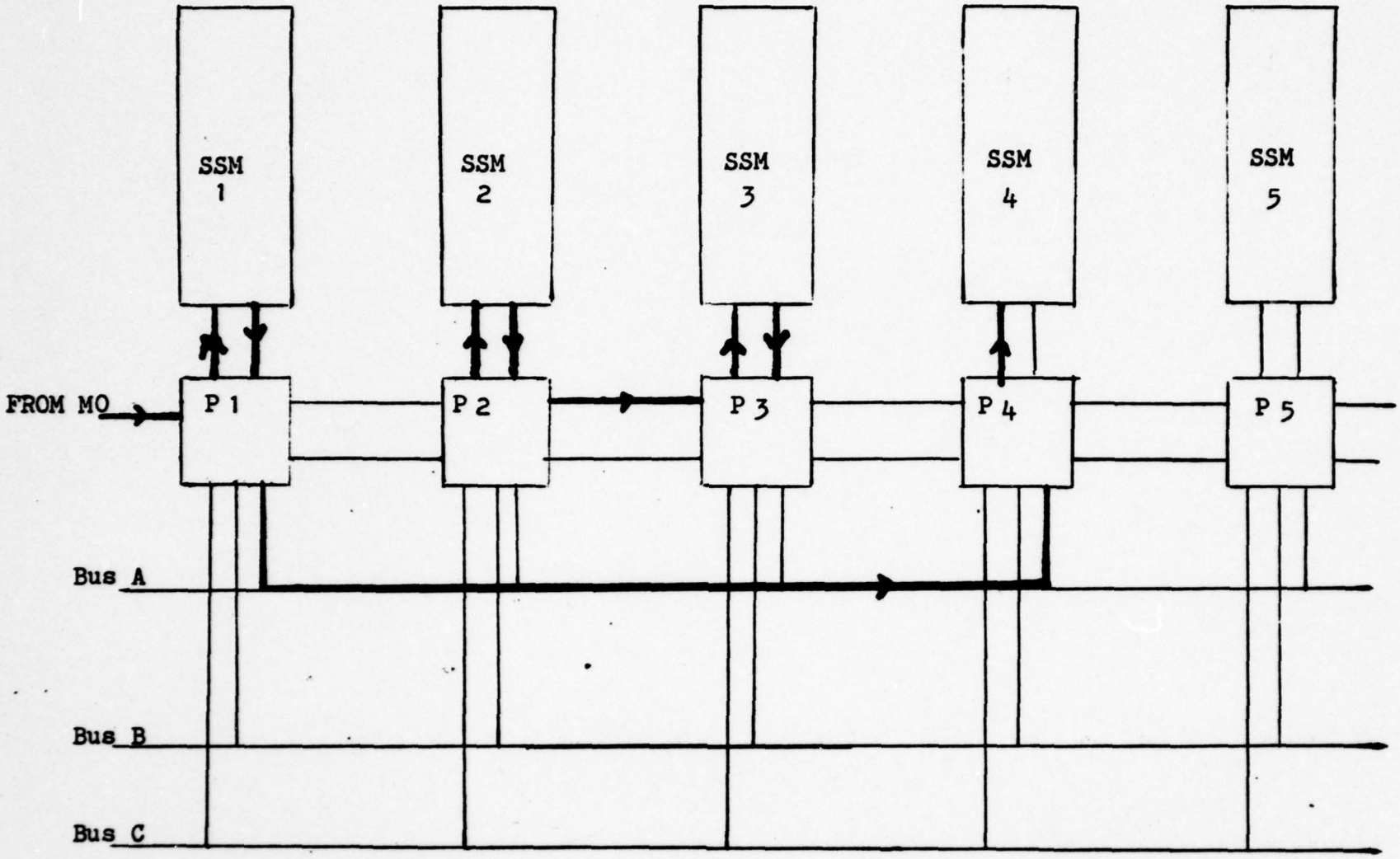


FIGURE 5

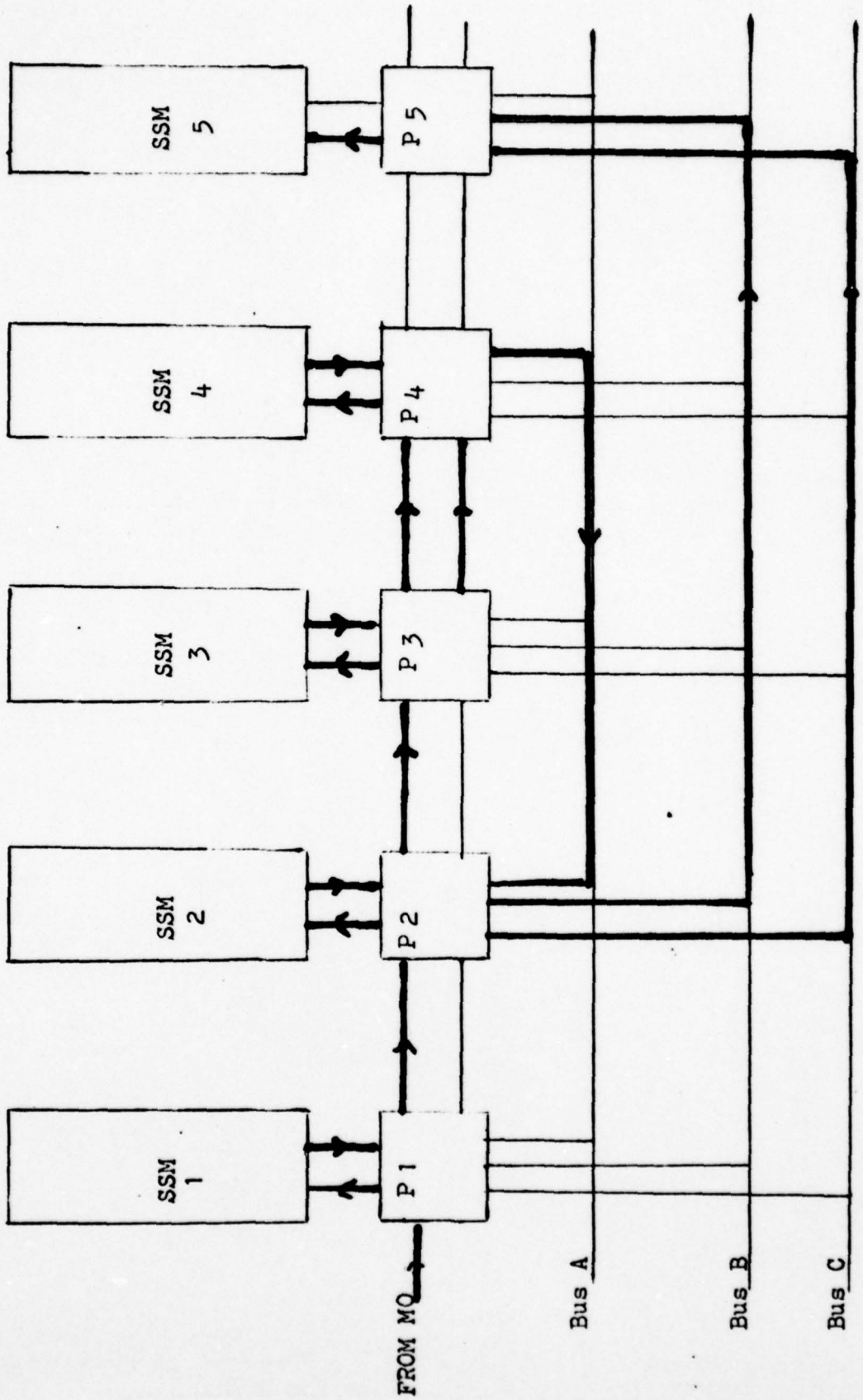


FIGURE 6

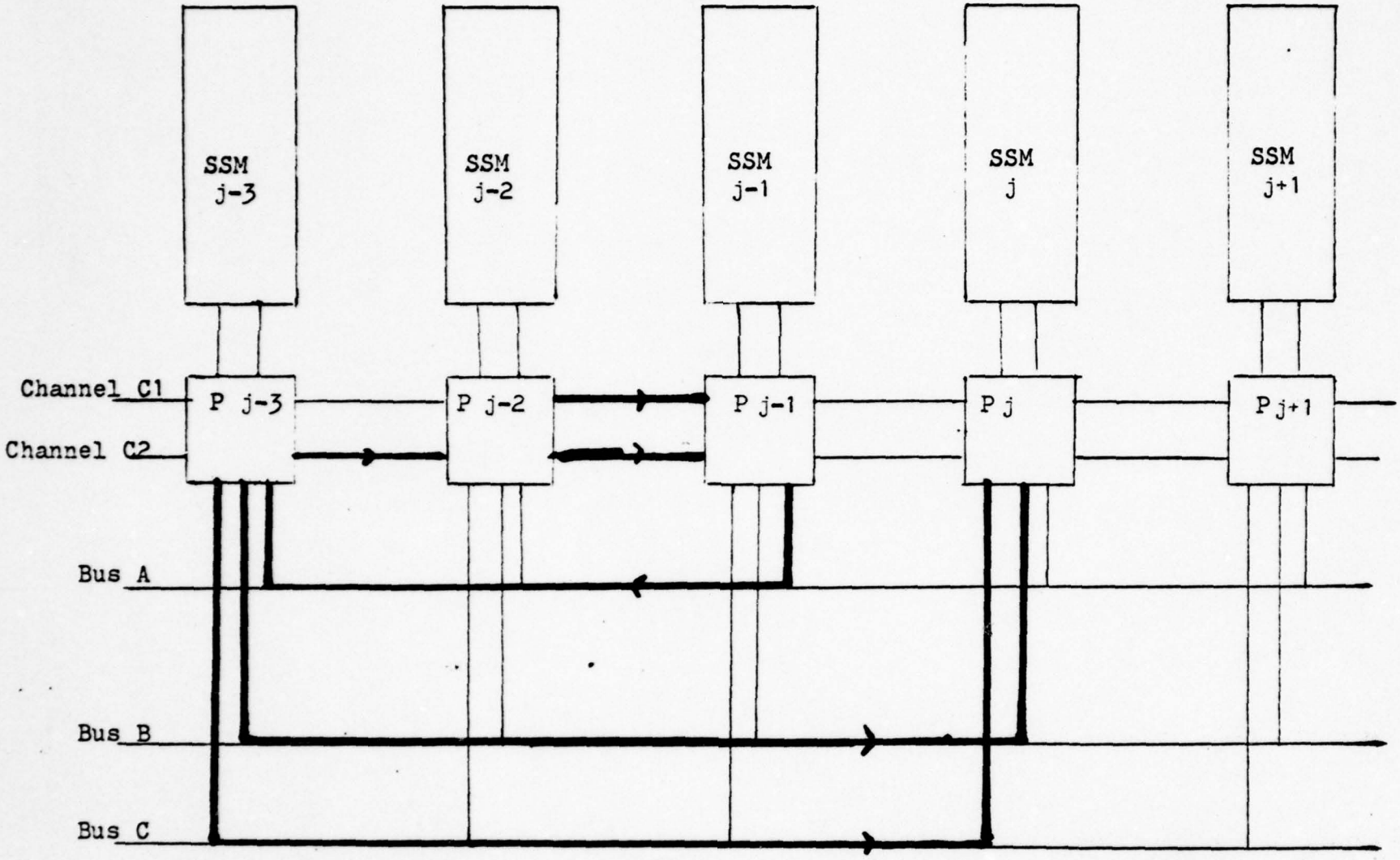
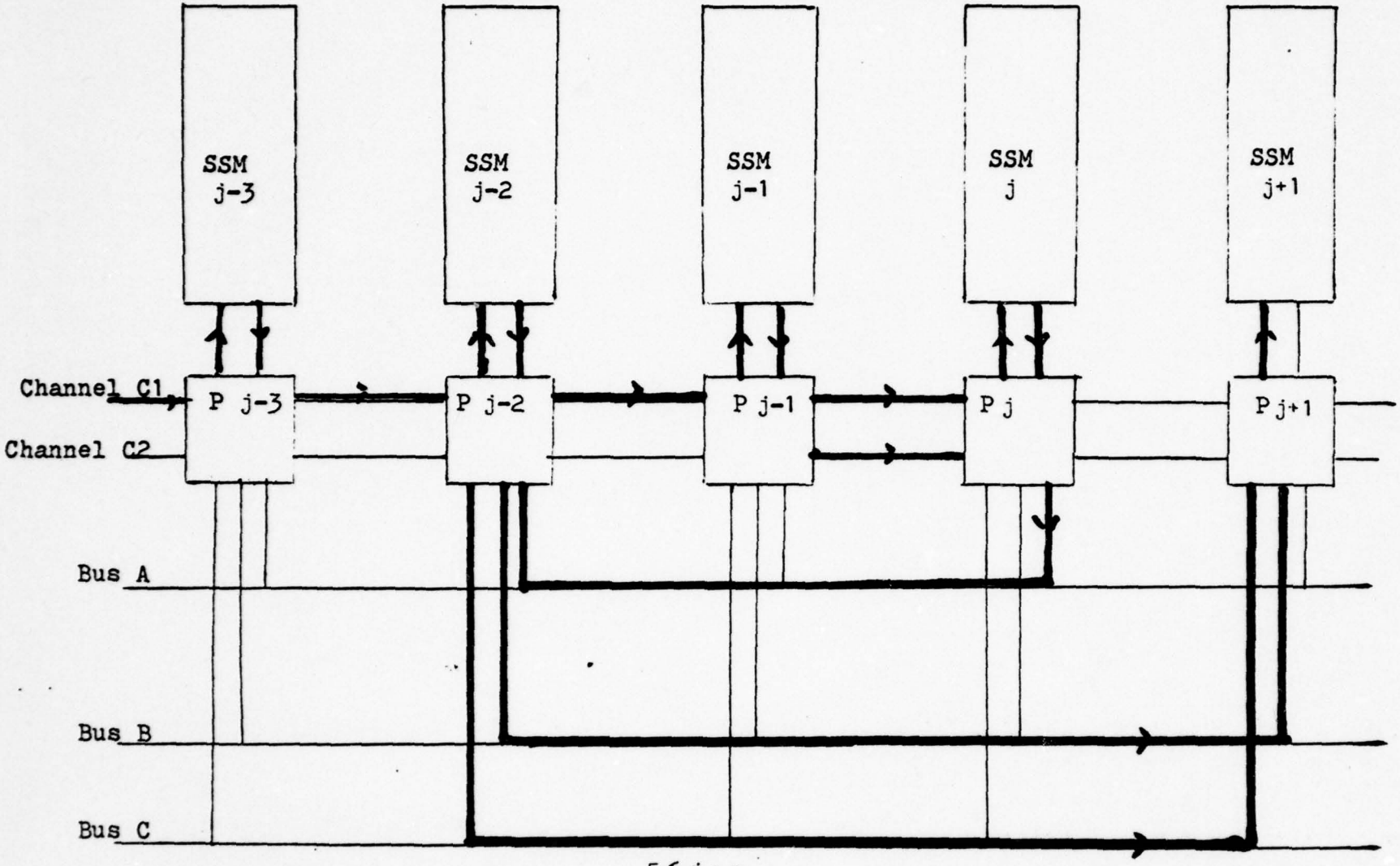

SSM j-3
SSM j-2
SSM j-1
SSM j
SSM j+1
Channel C1
Channel C2
P j-3
P j-2
P j-1
P j
P j+1
Bus A
Bus B
Bus C

FIGURE 7



$5 \leq j < n$

FIGURE 8

Operation Count And Timing

There will be required at least n^2 operations to transfer the matrix A to module M0 in System II. When the last column has passed to SSM0, its entries must pass through all of the processors P1, P2, ... , Pn and then to SSMn. This passage will require 2n operations, and after the entries are in SSMn (of course, one will be in Pn) the factoring computation will be complete. There will thus be a total of n^2+2n operations for this computation.

Solving For x

This procedure is composed of three parts: construction of the vector Pb, solving the system Ly = Pb for y, and finally solving the system Ux = y for x.

Computation of Pb

Matrix Ak, in the partial pivoting procedure, is obtained by assigning to each non-pivotal row i ($k < i$) the index i + 1 for those values of i for which $k \leq i < p_k$; the value i will not be changed outside this range. Then given the numbers p_k ($1 \leq k \leq n$) it is possible to reconstruct the permutation of the rows of the matrix A to obtain the matrix An. Thus as vector b is transmitted to P1, the component b1 will be retained in P1 if p_1 is 1; if the index in P1 is not 1, the index of b1 will be changed to 2 and will then pass to P2. When a component bi passes to P1, it will similarly have its index augmented if the number p_1 in P1 is greater than i. After b_{p_1} has passed to P1, the following entries will pass to P2 without change. Precisely this procedure will be followed in each processor so that at the time Pn receives a component of b it must have the index p_n. The result will then be the vector Pb, the components of which will be in the respective processors.

Solving for Y

The vector y is obtained from the recursive relation

$$y_i = b_i' - \sum_{j=1}^{n} m_{ij} y_j$$

(The component b_i' is the ith component of Pb; m_{ij} is the entry in the ith row of the jth column of L.)

In P1, the first component of y is obtained: $y_1 = b_1'$. This component is retained in P1 and simultaneously transmitted to P2, where y_2 is computed: $y_2 = b_2 - m_{21} y_1$. This procedure is followed in each processor--processor Pj first computes y_j and then combines the product $y_j m_{ij}$ with the sum R received from Pj-1 to obtain $R + y_j$, which Pj then transmits to Pj+1.

In the processor Pj, the entry m_{ij} is obtained from SSMj, of course. Since L is triagonal, Pj will be idle until row j is reached. Notice that this will be the first entry to appear from SSMj, since the entries of U will have mark bits M1 and M2 set to 1. Although at the beginning of the procedure all of the M1 bits will have been set to 0, the entries of U will still follow the entries of L because of the M2 bits.

At the end of this procedure, all of the components of y will have been computed and retained in the processors.

Back Substitution

The vector x may be obtained by computations of the kind described above, except the entries of the matrix U must be used and the columns of U must be scanned in the opposite order. Since the row indices of U will be smaller than the row indices of L in each column, access to the entries of U may be gained by setting the mark bits M2 to 0 after all of the entries of L have been transmitted to the respective SSM modules with M1 bit set to 1.

The computation, beginning in Pn, proceeds thus: processor Pn computes $x_n = y_n/u_{nn}$. The component xn is retained in Pn and is used to compute products of the form $x_n u_{in}$ which Pn transmits to Pn-1. In Pn-1, the component x_{n-1} is first computed: $x_{n-1} = y_{n-1} + \frac{x_n u_{n-1,n}}{u_{n-1,n-1}}$.

Processor Pj will compute $Q_j + x_i u_{ij}$, where Q_j is received in Pj from P_{j+1}, and will transmit this sum to Pj-1 until a processor Pi is reached, which will compute x_i:

$$x_i = \frac{Q_i + y_i}{u_{ii}} .$$

The vector x may be retained in the processors, or, as it is computed its components may be transmitted to the bus for output during the computation.

Timing

Computation of the vector x will require time for n operations after completion of the factoring process for obtaining the vector Pb, a second n operations for obtaining the vector y, and another n operations for obtaining the vector x. There will thus be 4n operations, including output of the vector x.

THE SIMPLEX METHOD

The problem for which the simplex method was formulated is the following:

Given:

1. Matrix $A = (a_{ij})$ $(1 \leq i \leq m,\ 1 \leq j < n;\ m < n)$

2. Row vector $a = (a_i)$ $(1 \leq i \leq n)$

3. Column vector $c = (c_i)$, $c_i \geqslant 0$ $(1 \leq i \leq m)$

Find the vector $x = (x_i)$ $(1 \leq i \leq n)$ such that

a) $c_i = \sum_{j=1}^{n} a_{ij} x_j$ $(1 \leq i \leq m)$

b) $\sum_{j=1}^{n} a_j x_j$ is minimum

c) $x_j \geqslant 0$ $(1 \leq j \leq n)$

There are, of course, many variations of this formulation which may be found in nearly any modern textbook on numerical methods. See, for example, the book <u>An Introduction To Numerical Mathematics</u> by Eduard L. Stiefel, published by the Academic Press. There are many references to pertinent literature in the publication <u>A Basis Factorization Method For Block Triangular Linear Programs</u>, Technical Report SOL 78-7, Stanford University, by Andre F. Perold and George B. Dantzig.

The method is defined on tableaux obtained from the entries in 1), 2), and 3) above. Following Stiefel's description, the system

1. $y_i = \sum_{j=1}^{n} a_{ij} x_j + c_i$ $(1 \leq i \leq m)$

2. $z = \sum_{j=1}^{n} a_j x_j$

is systematically changed by operations which exchange components y_i for x_j, thus finally obtaining positive values of the q entries x_j $(q \leq m)$ for which z is maximal. (The vector x is presumed to exist; the

problems of degeneracy do not present special data access problems and hence are not discussed in what follows.)

If the component x_j is exchanged for the component y_i, a new tableau is obtained which is computed according to the relations:

a) Replace a_{ij} by $-1/a_{ij}$

b) " a_{ik} by $-a_{ik}/a_{ij}$ $(k \neq j)$

c) " a_{kj} by a_{kj}/a_{ij} $(k \neq i)$

d) " a_{rs} by $a_{rs} - \frac{a_{rj}\, a_{is}}{a_{ij}}$ $(r \neq i,\ s \neq j)$

The computation may thus be composed of stages: at each stage, a selection of indices i and j will be made, and then in a succession of steps the above computations will be performed.

At the beginning of each stage, the selected pair of indices i and j will be exchanged. In an exchange operation, the entry c_i will be assigned the index j, and the column indices j will be changed to i. The indices initially assigned to the given entries c_i will be so that $1 \quad i \quad m$ and the indices j of x_j will be assigned so that $m+1 \quad j \quad n+m+1$.

At each step of the process, the entries a_{rs} will be computed. (The entry a_i will be the entry $a_{i,m+1}$ of the tableau and will be computed according to the specified rules a-d above. This procedure will also apply to the entries c_i, which will be treated as the entries $c_{i,n+1}$.) As the new entries a_i are computed, they will be compared so that the smallest one, the one for which $|a_i|$ is greatest and $a_i < 0$, may be selected. The index i of this entry will then be exchanged for the index j of the entry c_j for which $c_j/a_{ij} > 0$ is smallest, and $a_{ij} \quad 0$. The computation will terminate when all of the entries a_i are positive or when all of the quotients c_j/a_{ij} become negative. If all of the entries a_i are positive, there will be a solution, given by the entries which replaced the given entries c_i. More precisely the entries in the processors which replaced the entries c_i with indices greater than m will be the values of the corresponding components of x.

Mechanization

It is convenient to presume that the given entries a_{ij}, c, and a_i are in the store of System I. If they are not, transmission of these entries will proceed as in Gaussian elimination, except that a_{ij} will pass to SSMi (not SSMj). The SSM systems will then contain rows (not columns) of the matrix A. The entries $a_i = a_{m+1,i}$ will be in SSMm+1 and the entries $c_i = c_{i,n+1}$ will be retained in the processors Pi.

More precisely, the entries a_{ij} $(1 \leq j \leq n-2)$ will be in records with indices i and flag bits F1=0 in SSMi $(1 \leq i \leq m+1)$ and the two columns with entries $a_{i,n-1}$ and a_{in} will be retained in processors Pi $(1 \leq i \leq m+1)$. Processor Pm+1 may be used to store the number c, the maximal value of z.

The computation will begin by selecting the column j for which $a_{m+1,j}$ is minimal and negative. This is done by comparing the two entries $a_{m+1,n-1}$ and $a_{m+1,n}$ in Pm+1 and transmitting the larger one in its record with flag bit F1=1 to SSMm+1. This will displace the record from SSMm+1 with the least (column) index. The selection which is made in SSMm+1 will be transmitted to the other processors Pi via the system bus so that the remaining m records in the same column may also be transmitted to the respective SSM's with F1=1. The displaced entries will then pass to the processors and when n-2 records have passed from SSMm+1 with F1=1, the smaller of the two remaining entries will be the entry of the first pivot column.

After the first pivot column j is selected, the pivot row is identified by computing the quotients c_i/a_{ij}. That positive quotient which is the smallest will be the one with pivot index i so that a_{ij} will be the first pivot entry. This smallest quotient is obtained by transmitting from processor P1 the entry c_1/a_{1j} to P2. Processor P2 will compare c_2/a_{2j} with c_1/a_{1j} and transmit the smaller positive quotient to processor P3, etc. until finally the smallest positive quotient is passed to Pm+1. Of course if in some processor P_i the quotient is negative, it will simply transmit what it received from Pi-1. If no positive quotient is passed to Pm+1, the procedure will terminate.

This selection of the pivot row will require the time to send a record from P1 to Pm+1. This selection is done once for each column scan of the n entries, or once for each stage.

When the pivot entry a_{ij} is identified, the arithmetic computation may begin. The quotients $-a_{kj}/a_{ij}$ $(i \neq k)$ and $-1/a_{ij}$ are formed and retained in the processors Pi. The entries of the second (non-pivotal) column s which is also in the processors may now be replaced by the entries computed in this first step: a_{rs} will be replaced by

$$a_{rs} - \frac{a_{rj} a_{is}}{a_{ij}}$$

The entry a_{rs} will be found in processor Pr. The entry $-a_{rj}/a_{ij}$ will be in the pivot processor Pi. Thus as the entries of column s are computed, pivot processor Pi will transmit to each processor Pr $(1 \leq r \leq m)$ the entry a_{is} for computation in Pr of the entry a_{rs}. The entry a_{is} will be transmitted via the system bus, of course.

The computed entries a_{rs} will be placed in records with the key s and with flag bits F1 set to 1. The records will then be transmitted to SSM storage for displacement of the column records with least indices for which F1=0.

This procedure will be repeated until finally the (new) pivot column itself is transmitted to SSM storage.

As the successive entries $a_{i,m+1}$ are computed in Pm+1 they will be compared in this processor so that when all of the entries $a_{m+1,i}$ $(1 \leq i \leq n)$ have been computed, the new smallest one will be identified and retained, with the remaining m entries of its column, in processors Pi $(1 \leq i \leq m+1)$.

Notice that the sequence of the columns which are scanned or computed is of no significance: an entry of a column may be assigned any (unique) key without changing the procedure. Thus when an entry a_{ij} is selected as the pivot, its column key will be changed to i and the entry $a_{i,m+1}=c_i$ will be assigned the key j. (The entries c_i will be retained in the processors.)

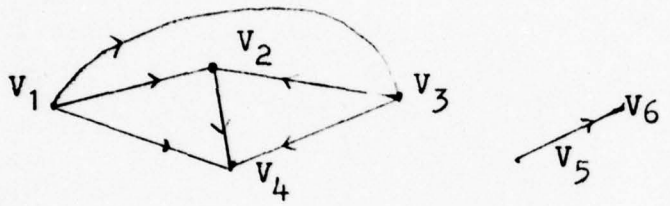
When it is found that all of the entries $a_{i,m+1}$ are positive, the procedure will terminate. The desired values of x are then the entries c_i for which $i > m$.

Operation Count And Timing

The number of operations in a stage will be the number n (i.e. the number of entries in a row of the matrix A) plus the number of operations needed to select the least entry c_i/a_{ij} of the m computed quotients. If the processors are equipped to compare and conditionally exchange records, the time needed for selection of the least of the m quotients should not be more than the time needed to pass a record from processor P1 to Pm. This should be approximately the amount of time needed for m operations. Thus the time for a stage should be approximately, and not more than, the time needed for n+m operations. A conventional computer will require an amount of time proportional to $tn(m+1)+(m+1)^2$. (See the book Linear Programming and Extensions by George B. Dantzig, published by the Princeton University Press, 1963, page 216.)

THE TRANSITIVE CLOSURE OF A GRAPH

A graph, as that term is used here, is simply a set with a relation defined over some of its elements ("vertices"). A city street map with one-way streets is a fair illustration of the idea. Some vertices may be connected to others by a relation for which the inverse does not exist. But some of the vertices may be related to no others. Thus, for example, the six vertices shown may represent a graph.



Such a graph is said to be a directed graph because if V_1 is related to V_2, indicated by the line ("edge") from V_1 to V_2, V_2 may not be related to V_1 (whether such a relation exists is indicated by the arrows on the lines in the diagram). Notice that V_5 is related to V_6 but to none of the other vertices. V_2, V_3 and V_4 are related, but not to V_1, although V_1 is related to V_2, V_3, and V_4.

There are many applications of the theory of graphs--such a simple idea will clearly underlie almost any theory of networks, for example. The reader is referred to the book New Directions In The Theory of Graphs edited by Frank Harary, published by the Academic Press (1973) for a description of the theory and for other references.

Let A be the incidence matrix of the directed graph G with n vertices. It is desired to obtain the matrix $B = (b_{ij})$ such that if there is a non-trivial path from vertex V_i to vertex V_j, $b_{ij} = 1$

and otherwise $b_{ij} = 0$. A path is called nontrivial if it contains at least one arc. If we allow trivial paths as well, all elements on the main diagonal of B will be 1. This closure, with trivial paths, is called the transitive-reflexive closure; the transitive closure is defined by the matrix B.

The concept of a transitive closure occurs in many different places. The question, e.g., whether the graph G contains a directed cycle is equivalent to the question whether B has a 1 on the main diagonal. The question whether vertex V_i and vertex V_j are members of the same strong component of G is equivalent to the question whether $b_{ij} = b_{ji} = 1$.

We mechanize Warshall's algorithm (see Aho, Hopcraft and Ullman, The Design And Analysis of Computer Algorithms, Addison-Wesley, 1974). The program consists of n steps. After step k ($0 < k \leq n$) the following relation will hold.

P $b_{ij}=1$ if and only if *there exists a nontrivial path* from vertex V_i to vertex V_j with all internal vertices chosen from the set $V_1, V_2, - - - V_k$

Obviously, for k=n B has the desired value. For k=0, the initial state, B is equal to A. In step k+1 we apply the following operation to all elements b_{ij}:

S $b_{ij} := b_{ij} \vee (b_{i,k+1} \wedge b_{k+1,j})$,

which maintains the validity of relation P.

We employ System I. Module i will contain the elements of column i. We execute operation S by broadcasting column k+1 to all other modules. Each module j ($j \neq k+1$) changes element b_{ij} only if b_{ij} was 0, the $b_{i,k+1}$ broadcast is 1, and $b_{k+1,j}$ is 1. First module j checks whether $b_{k+1,j}$ is 1. If so, it runs through its column concurrently with the broadcasting of column k+1. This can be done

in n time steps. Therefore the whole closure process requires n^2 steps. If we require only the transitive-reflexive closure, we simply start out with all elements b_{ii} equal to 1.

If we wish to determine the number of different non-trivial paths between every two vertices, we replace the operation S by

$$b_{ij} := b_{ij} + (b_{i,k+1} \wedge b_{k+1,j}),$$

in which the "+" denotes integer addition.

Another variant to the scheme above is the computation of the shortest paths between every pair of vertices. Let A be the arc length matrix, i.e. a_{ij} is the length of the arc from vertex V_i to vertex V_j. All arc lengths are nonnegative. If there is no arc from V_i to V_j, then a_{ij} has some very large value. We can compute the length of a shortest path for every pair of vertices by replacing the operation S by

$$b_{ij} := \min(b_{ij}, b_{i,k+1} + b_{k+1,j})$$

The shortest paths computation can thus be done in $O(n^2)$ time. This computation requires $O(n^3)$ time for a sequential machine. With n processors the complexity is decreased by a factor of n.

Notice that if the format of the records is changed as the last column is completed, the columns may be sorted on the distance between vertices. Those which are closest will appear first in their columns and will be followed by the entries with largest distances.

It should be remarked that for an nxn matrix we presume n processors to reduce the number of steps from n^3 to n^2. It may well be that not all of the processors will be "busy". If, for example, there is a vertex V_i which is not connected to any others, processor P_i will have nothing to do at all. There is no apparent way to make an exact count of the required number of operations: we count the maximal number, or the maximal amount of time which will be consumed.

A Serial Sorting Machine.

Philip N. Armstrong [+])
Martin Rem [++])

Abstract.

A simple sorting machine consisting of serial storage elements and comparator modules is discussed. It sorts a sequence in the time required for input and output; no additional time for internal sorting is required. It is shown how the machine can be augmented to perform various types of file manipulating tasks.

C.R. Categories.

3.7, 3.73, 3.74, 5.31, 6.22, 6.34, 6.36.

Key words and phrases.

sorting, sorting machine, serial sorting machine, self-sorting memory, serial storage, comparator module, searching, file manipulation, associative addressing.

[+]) 17331 Keegan Way
Santa Ana, CA 92705

[++]) Department of Computer Science
256-80 California Institute of Technology
Pasadena, CA 91125

After July 1, 1978:
Department of Mathematics
Eindhoven University of Technology
P.O. Box 513
Eindhoven, The Netherlands

PREPARATION OF THIS DOCUMENT WAS SUPPORTED BY THE OFFICE OF NAVAL RESEARCH UNDER CONTRACT N00014-78-C-0357.

A Serial Sorting Machine.

Philip N. Armstrong and Martin Rem

1. Introduction.

We discuss a simple machine that sorts a collection of items in the time required for input and output of the items.

The machine is composed of serial storage elements (shift registers or delay lines) and special sort circuits or comparator modules. It is a serial machine: the items to be sorted are fed serially into the machine at the bit rate of the storage elements. Immediately after the last item has been inserted we can extract the items in sorted order; no additional time for sorting is required, regardless of the capacity of the machine. With a serial machine this is the best one can achieve. (Notice that we do not necessarily claim that our machine will contain the sorted arrangement: we only say that the items may be withdrawn in sorted order.)

Sorting machines composed of sorting and storage elements are described by D.E. Knuth [5]. Those acquainted with [5], particularly the discussion on page 244, will recognize our machine as one with "moving heads" wherein the heads scan adjacent record positions. The type of comparator modules outlined in this paper, together with networks of such modules, were first described by R.J. Nelson, c. 1954 at the IBM corporation (see [1]). A related, but apparently more complicated, machine is mentioned in [4]. Some of the properties of networks of comparator modules are contained in [3]. Descriptions of several facets of the machinery to be described are contained in [2].

2. Description of the machine.

In essence the sorting machine is a storage medium with a fixed word size w storing, in binary representation, a fixed number N of natural numbers. Among the numbers in the store there is a least one and a, not necessarily distinct, greatest one; these numbers, of course, may be multiply present. There are only two operations that can be performed on the store:

1. Replace a least number by a greater number,
2. Replace a greatest number by a lesser one.

The machine consists of $N + 1$ compare/exchange ("comparator") modules and $2N$ serial storage elements. They are interconnected as in Fig. 1; for simplicity we have left out wires for ground, power, and clock.

(Fig. 1)

The arrows denote wires that convey bits, the arrow heads denoting the direction of conveyance. The vertical boxes denote the comparator modules. Each of them has two inputs: one from the left, the "low input", and one from the right, the "high input". The two outputs are similarly called the "low output" (to the left) and the "high output" (to the right). A comparator can be in any one of three states: undetermined, back, or through. In the back state the low and high inputs are connected to the low and high outputs, respectively. In the through state the two outputs are interchanged: the low and high inputs are connected to the high and low outputs, respectively. In the undetermined state either of these two connections may be chosen.

The comparator module goes into the undetermined state by means of an external "begin signal" (not shown). This signal is given every w bit times. The comparator receives a pair of bits, one bit from each input, at a time and remains in the undetermined state until its two input bits differ, upon which occasion it goes into the back state if the low input is 0, and into the through state otherwise. It does not leave the back or through state until a new begin signal is applied to it. Fig. 2 shows a possible realization of a comparator module. There are many possible variations of this circuit; the one shown is intended merely to indicate its lack of complexity and is undoubtedly not the simplest possible circuit.

(Fig. 2)

The horizontal boxes in Fig. 1 denote the serial storage elements. There are two types: upper registers moving the bits to the right, and lower registers that move their contents to the left. Denoting the capacity of a

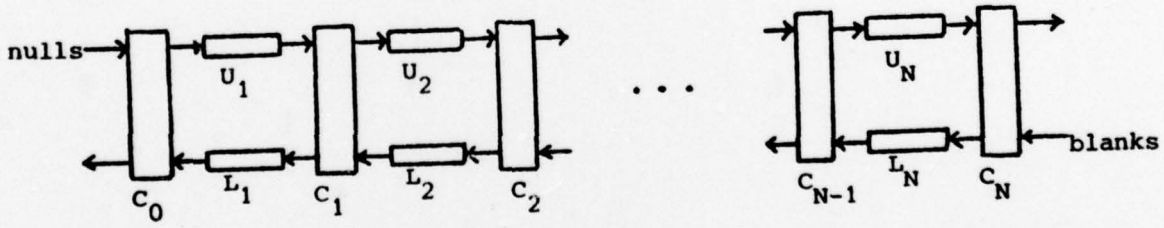
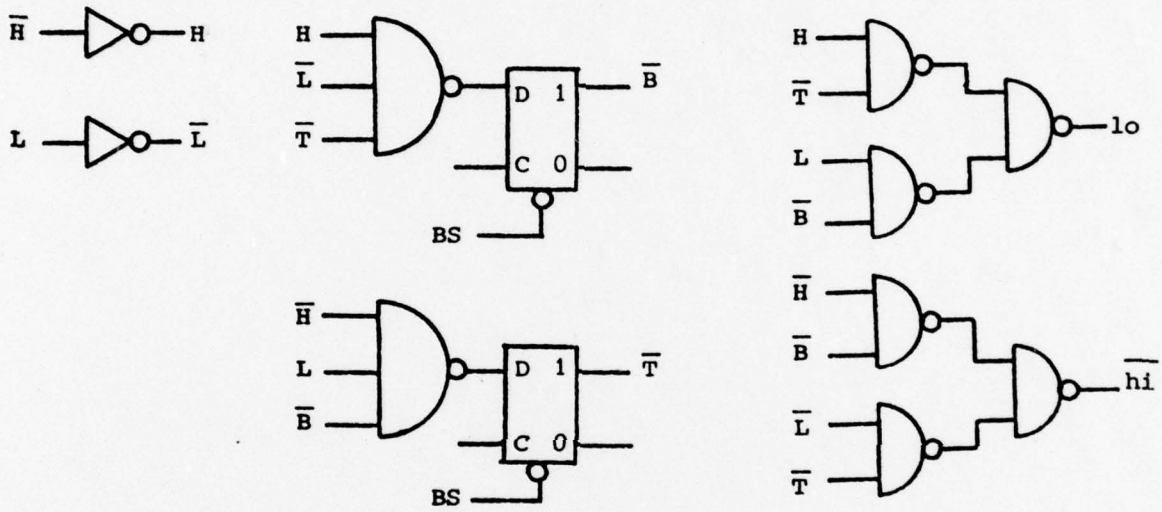


Fig. 1



L	low input	
H	high input	
T	through state	
B	back state	
lo	low output	$= H.\overline{T} + L.\overline{B}$
hi	high output	$= (H + B).(L + T)$
BS	begin signal	

Fig. 2

register R by $l(R)$, the registers satisfy

$$l(U_i) > 0 ,$$
$$l(R_i) > 0 ,$$
$$l(U_i) + l(R_i) = w .$$

We call registers U_i and R_i together "ring i ". There are N rings; each ring has the capacity to store one number.

The low input and output of C_0 are called the low input and output of the machine. We likewise call the high input and output of C_N the high input and output of the machine. A "null" is a series of w zeroes, a series of w ones is called a "blank". We feed a continuous stream of nulls at the low input of the machine; they immediately leave the machine via its low output. Similarly we feed blanks to the high input. The numbers that are stored in the machine remain there. In the next section we show that they will be put into sorted order.

3. The machine sorts its contents.

A number consists of w bit positions, numbered 0 through $w - 1$. Let position 0 be the position of the most significant bit. The numbers are placed in the store in such a way that the comparators process the bits in the order of decreasing significance.

The comparator modules operate in strict synchronism: each comparator Processes one pair of bits per step. Such a step keeps the following two relations invariantly true.

(1) Going counterclockwise through a ring one encounters bit positions $0,1,2, \ldots ,w-1,0, \ldots$, not necessarily belonging to the same number.

(2) The two inputs of any comparator module contain corresponding bit positions.

The invariance follows immediately from the fact tnat a step replaces each bit position j in a register by a bit position $(j + 1) \bmod w$.

We say that a comparator is in position i when both inputs are bit positions i . As a consequence of invariance (1), the comparators go

cyclicly through the positions $0, 1, 2, \ldots, w - 1$. The begin signal is applied upon the comparator's entering position 0, putting the comparator into the undetermined state. While the comparator is in this state its two outputs will be equal to each other. As soon as the inputs differ the comparator goes into the back or into the through state, sending the remainder of the greater number over the high output and that of the lesser number over the low output. A comparator, consequently, cannot scramble a number: it always transmits a complete number via the same output.

A number is said to be in ring i if its bit position 0 is in ring i. The truth of (1) implies that every ring contains exactly one number. We say that there is an "inversion" between rings i and j if $i < j$ and the number in ring i is greater than that in ring j. The number of inversions is at most $\frac{1}{2}N(N - 1)$. The store is considered to be sorted if there are no inversions. If the store is not sorted there must be a comparator C_i such that there is an inversion between rings i and $i + 1$. This comparator will exchange the numbers in these rings during its next cycle, thereby decreasing the number of inversions. A comparator will only make an exchange if it has detected an inversion. Eventually a state must be reached without inversions.

Remark: We said that the comparator modules operate in strict synchronism. It is true that they process pairs of bits in synchronism; they do not, however, process numbers in synchronism: the position of C_i is equal to the sum, taken modulo w, of the position of C_{i-1} and $l(L_i)$. This relation determines the moments at which the begin signals must be applied to the comparators.

We can now describe how to replace a number in the machine by a different number. Replacement of a least number by a greater one is accomplished at the low end of the machine: when a least number is in ring 1 and C_0 is in position 0 we apply the new number at the low input; if it is greater than the number in ring 1 it will repace that number. Afterwards the least number is again in ring 1 and we can immediately replace it again by a greater number. There is no delay between these operations. The other operation, the replacement of a greater number by a smaller one, is carried out analogously at the high end of the machine.

There is an inherent delay if we wish to switch between the two types of operations: we must give the newly inserted number the opportunity to travel through the store. This delay is $\Sigma_{i=1}^{N} l(U_i) - w$ bit times when switching from low insertions to high insertions and $\Sigma_{i=1}^{N} l(L_i) - w$ when switching in the other direction. In section 5 we will present an alternative architecture that does not have these delays.

4. Applications of the sorter.

In this section we discuss some applications of the machine, including of course sorting. We assume that the word size w is sufficient to accommodate the numbers involved. In section 5 we will show how to process multi-length items.

We can fill the machine with any initial N, not necessarily mutually distinct, numbers by applying N nulls at the high input of the machine, followed by applying the N required numbers at the low input. Each number is coded in w bits; we always insert the bits in the order of decreasing significance.

Sorting a sequence of k ($k \leq N$) elements.

Let it be requested to sort a sequence of k, not necessarily distinct, natural numbers into nondecreasing order. We fill the machine initially with k nulls and $N - k$ blanks. We then apply the sequence at the low input, immediately followed by k blanks. During the application of these blanks the ordered sequence will leave the machine via its low output.

For sorting into nonincreasing order we fill the machine initially with k blanks and $N - k$ nulls and apply the sequence followed by k nulls at the high input, thereby producing the sequence in nonincreasing order at the high output. In both cases no additional time for internal sorting is required.

Determining k ($k \leq N$) extreme elements of a sequence of arbitrary length.

This task is very similar to sorting a sequence of length k. If we wish to find k greatest numbers of a sequence we fill the machine initially

with k nulls and $N - k$ blanks. As in the case of sorting we then apply at the low input the sequence followed by k blanks. During the application of the blanks the k greatest numbers are, in sorted order, produced at the low output. One can likewise generate the k least numbers at the high output.

We can determine the median of a sequence of $2k - 1$ numbers by applying at the low input the sequence followed by one blank. As the blank is fed into the system the median of the sequence comes out.

Other items than natural numbers.

For simplicity we have described our machine as a sorter for natural numbers in binary representation, stored in such a way that the comparators encounter their bits in the order of decreasing significance. There is no reason to restrict ourselves to natural numbers. If we wish to accommodate negative integers as well, the numbers should be coded in a variant of one's or two's complement notation in which the least significant bit is inverted and attached to the other end of the word. Floating point numbers may be accommodated if they are represented in some normalized form, e.g., with their exponents having minimal absolute values. The exponent part should, of course, be treated as the most significant part of the representation. We can also sort records each consisting of a key and a data part; the key should again be treated as the most significant part.

Rearranging the contents of the machine.

If we have keys attached to the items in the machine we can rearrange their order by changing the keys. As the items come out of the store we can reinsert them with different keys. For some simple rearrangements we do not need keys; they can be effected by changing the moments at which we apply the begin signals to the comparators. For example, we can have the contents of the store circulate by extending each item in the store with two "mark bits": one, equal to 1, at the least significant end and one, equal to 0, at the most significant end. We perform our input and output at the low end of the machine. As each item comes out of the store we put it

back with its 1-mark bit appended to its most significant end. After a complete circulation we apply to each comparator the begin signal one bit time later than usual, which effectively puts the 1-mark bits back at the least significant ends, thereby conditioning the store for another circulation of its contents.

By changing the timing of the begin signals we can also perform a sort over different keys. Suppose we have in our machine records, each consisting of a data part and three keys. For any of the three keys we wish to be able to scan the records in the order of increasing key values. Let each record consist of, with increasing significance, a 1-mark bit, a data part, key_1, a 0-mark bit, key_2, a 0-mark bit, key_3, and another 0-mark bit. With the above technique we can scan the records for increasing values of key_3. If we wish to scan the records for increasing values of, say, key_1 we append key_2 and key_3 and their trailing 0-mark bits to the least significant end of each record by appropriately delaying the begin signals. Then we extract the records for increasing values of key_1, putting them back in their original formats with the 1-mark bits appended to the most significant ends. Of course, there is a delay before we may begin the extraction. If we choose $l(L_i) = 1$ this delay is N bit times.

5. Extensions to the machine.

Multi-length items.

We have assumed that the word length w was sufficient to represent the items. We can, however, also use the machine to process items whose lengths are multiples of w. Suppose all items have length $k \times w$, for some fixed $k > 0$, then we can effectively multiply by k the sizes of the rings by fixing all comparators C_i for which $i \bmod k \neq 0$ in the through state. We can achieve this by explicitly putting these comparators into this state and never applying a begin signal to them. It seems, however, simpler to have the comparators check whether they are processing the first bits of two items. We do this by fixing the values of k bits in each item: for each item bit position 0 (the most significant bit) is always 0, the other bit positions i for which $i \bmod w = 0$ are always equal to 1. We change each comparator in such a way that it enters the undetermined state only when the begin signal is applied to it and both of its inputs are 0.

Abolishing the delay.

Our machine has essentially two operations: replace a least number by a greater number, and replace a greatest number by a lesser one. As we have pointed out earlier, switching from the one type of operations to the other requires a delay. One encounters this delay, for example, if one wishes to interleave insertions (replacements of blanks) with deletions of least numbers (replacements by blanks). We can augment our machine in such a way that we can perform both input and output at the same end of the machine, thereby abolishing these delays. There are a number of augmentations that accomplish this; we discuss one that uses N additional comparators and N additional serial storage elements. The resulting machine for input and output at the low end, i.e., with sorting into nondecreasing order, is shown in fig. 3.

(fig. 3)

The added registers A_i satisfy $l(A_i) = l(U_i)$. The added comparators Z_i, like the other ones, send the greater number to the right and the lesser one to the left. The machine has two input channels called the top input and the bottom input, respectively. Unless stated otherwise we apply nulls at the top input and blanks at the bottom one. All registers A_i, consequently, contain blanks; as a result the comparators Z_i are always in the through state and do not affect the functioning of the machine.

As in the basic machine we replace a least number by applying a greater number at the top input. A greatest number is replaced by applying a lesser one at the bottom input. As all input is performed at the left there will always be a least number in ring 1. We still have to check that the application of a number at the bottom input does indeed displace a greatest number. The critical case is when there is one greatest number that has been inserted, via the top input, during the immediately preceding operation. When we apply the number at the bottom input the greatest number will have traveled w bit positions to the right. It follows the path $U_1, U_2, \ldots, U_N, L_N$, and will thus be at the high input of comparator Z_{N-1} after

$$\Sigma_{i=1}^{N-1} l(U_i) + l(U_N) + l(L_N) - w = \Sigma_{i=1}^{N-1} l(U_i)$$

bit times. The number inserted at the bottom input travels the path $A_1, A_2, \ldots$ until it meets a lesser number; it is interchanged with that number and the new number continues the path along the registers A_i until it in its turn meets a lesser number, etc. After $\Sigma_{i=1}^{N-1} l(A_i)$ bit times the original or some lesser number reaches the lower input of comparator Z_{N-1}, where, as $\Sigma_{i=1}^{N-1} l(A_i) = \Sigma_{i=1}^{N-1} l(U_i)$, it meets the greatest number, causing the greatest number to leave the machine.

Remark: This scheme works for multi-length items as well.

Remark: We can trivially use the machine of fig. 3 to sort into nonincreasing order by inverting the inputs and the outputs.

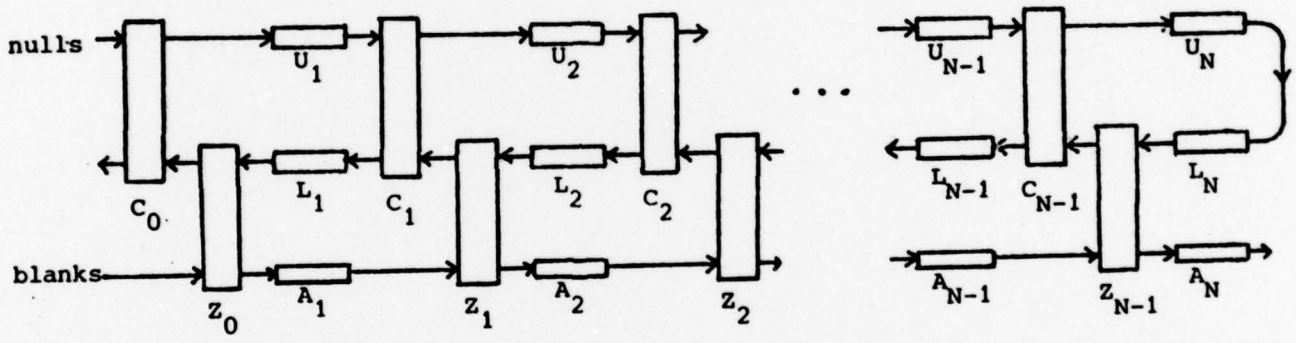

nulls
blanks
U_1
U_2
U_{N-1}
U_N
L_1
L_2
L_{N-1}
L_N
C_0
C_1
C_2
C_{N-1}
Z_0
Z_1
Z_2
Z_{N-1}
A_1
A_2
A_{N-1}
A_N

Fig. 3

Associative updating and retrieval.

We can augment the machine above to accommodate associative addressing. The store contains records, each record consisting of a key and a data part. The comparators Z_i are equipped with switches by which they can be put into a mode in which they detect equality only. In this mode a comparator is in the through state until it has passed a complete key; at that moment it goes into the back state if and only if the two keys compared were equal.

We may start the associative operations as soon as the store's contents are fully sorted. We can then perform an update of a record by applying a new record with the same key at the bottom input. If the key is absent nothing happens; eventually the new record will leave the system via the high output. If the key is present the record with that key will be replaced by the new one. These associative updates may be pipelined: a sequence of m updates can thus be processed in $\Sigma_{i=1}^{N} l(A_i) + m \times n$ bit times.

To permit associative retrieval as well we must equip the channels with switches (fig. 4).

(fig. 4)

With the switches in position 2 and the comparators Z_i in the mode to test equality we apply a record consisting of the key and some dummy data part at the bottom input. If the key is absent the record will leave the system again via the high output; if it is present the associated data part will be copied into the record's data part, which will come out via the high output. These retrieval operations may also be pipelined. It will be apparent that numerous variations to this scheme are possible.

6. Conclusions.

The essential ideas of the machine are simplicity and economy; the combination of serial storage elements and comparator modules permits rapid sorting without the intervention of a high speed computer. With a bit time of 50 ns , which does not seem unreasonable for storage elements like MOS shift registers or charge-coupled devices, 150,000 64-bit numbers can be sorted in less than a second.

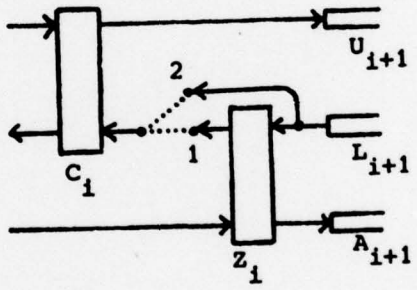


Fig. 4

It turns out that such a machine can easily be augmented to perform other file manipulations as well. As an example we considered associative updating and retrieval.

7. References.

(1) Armstrong, P. N., Marcus M.P., and Nelson, R. J. *U.S. Patent 3034102.* May 8, 1962.

(2) Armstrong, P.N. *U.S. Patents 3329938, 3329939, 3336580, 3587057, and 3399383.* 1978 - 1971.

(3) Batcher, K.E. Sorting networks and their applications. *Proc. AFIPS Spring Joint Computer Conference 32* (1968), 307 - 314.

(4) Edelberg, Murray and Schissler, L. Robert. Intelligent memory. *Proc. AFIPS National Computer Conference 45* (1976), 393 - 400.

(5) Knuth, Donald E. *The Art of Computer Programming Vol. 3.* Addison-Wesley, Reading, Massachusetts, 1973.